ELSEVIER The Journal of Logic Programming 36 (1998) 149–193

THE JOURNAL OF LOGIC PROGRAMMING

Creating specialised integrity checks through partial evaluation of meta-interpreters

Michael Leuschel *, Danny De Schreye [1]

Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium

Received 1 May 1996; received in revised form 1 October 1997; accepted 24 November 1997

Abstract

Integrity constraints are useful for the specification of deductive databases, as well as for inductive and abductive logic programs. Verifying integrity constraints upon updates is a major efficiency bottleneck and specialised methods have been developed to speedup this task. They can, however, still incur a considerable overhead. In this paper we propose a solution to this problem by using partial evaluation to pre-compile the integrity checking for certain update patterns. The idea being, that a lot of the integrity checking can already be performed given an update pattern without knowing the actual, concrete update. In order to achieve the pre-compilation, we write the specialised integrity checking as a meta-interpreter in logic programming. This meta-interpreter incorporates the knowledge that the integrity constraints were not violated prior to a given update. By partially evaluating this meta-interpreter for certain transaction patterns, using a partial evaluation technique presented in earlier work, we are able to automatically obtain very efficient specialised update procedures, executing faster than other integrity checking procedures proposed in the literature.

1. Introduction

1.1. General context

Partial evaluation has received considerable attention both in functional programming (see e.g. [18] or [37] and the references therein) and logic programming (e.g. [41,42,31,30,83]). However, the concerns in these two approaches have strongly differed. In functional programming, self-application and the realisation of the different Futamura projections, has been the focus of a lot of contributions. In logic program-

* Corresponding author. Tel.: +32 16 327 555; fax: +32 16 327 996; e-mail: michael@cs.kuleuven.ac.be.
[1] E-mail: dannyd@cs.kuleuven.ac.be.

0743-1066/98/$19.00
PII: S0743-1066(97)10012-7

ming, self-application has received much less attention. [2] Here, the majority of the work has been concerned with direct optimisation of run-time execution, often targeted at removing the overhead caused by meta-interpreters.

In the context of pure logic programs, partial evaluation is often referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of non-declarative programs. Firm theoretical foundations for partial deduction have been established by Lloyd and Shepherdson in [60].

A *meta-program* is a program which takes another program, the *object program*, as input and manipulates it in some way. A detailed account of meta-programming and its uses can be found in [35]. Some of the applications of meta-programming are e.g.: extending the programming language, multi-agent reasoning, debugging, program analysis and transformation.

From a theoretical viewpoint, *integrity constraints* are very useful for specifying deductive databases as well as inductive or abductive logic programs. [3] They ensure that no contradictory data can be introduced and monitor the coherence of a database. From a practical viewpoint, however, it can be quite expensive to check the integrity of a deductive database after each update. To alleviate this problem, special purpose integrity simplification methods have been proposed (e.g. [11,21,63,61, 81,19]), taking advantage of the fact that the database was consistent prior to the update, and only verifying constraints possibly affected by the new information. However, even these refined methods often turn out to be not efficient enough.

1.2. Motivation and contribution

There have been two lines of motivation for this work. A first motivation came from our participation in the ESPRIT-project Compulog. This project brought together European research groups, both from the areas of deductive databases and logic programming. One idea, that was a frequent issue of discussion among the partners from the different areas in the project, was the potential of deriving highly specialised integrity checks for deductive databases, by *partially evaluating* refined integrity checking procedures, implemented as *meta-interpreters*, with respect to particular update patterns and with respect to the more static parts of the database.

This idea is appealing, because the application is very well suited for partial evaluation. The selected technique for performing the integrity checking is a static component of the application, as well as the different potential update patterns which may occur. Often, the actual contents of the stored database relations (the extensional database – EDB) is rather dynamic, while the rules defining derived relations from stored ones (the intentional database – IDB) are more static. Due to this possibility of dividing the application into two larger parts, one of which is static, the other dynamic, the potential of gaining speedups by specialising the static part with respect to a number of different patterns (often just the relational symbols) for the dynamic part seemed extremely promising.

[2] Some notable exceptions are [71,33,34,39].

[3] In the remainder of this paper we will only talk about deductive databases, but the discussions and results remain of course also valid for inductive or abductive logic programs with integrity constraints.

Moreover, this would allow for a very flexible way of generating specialised update procedures. Any kind of update pattern and any kind of partial knowledge could be considered. For instance, in [12], Bry and Manthey argue that for some applications facts change more often than rules and rules are updated more often than integrity constraints. As such, specialisation could be done with respect to dynamic EDBs, as well as with respect to dynamic IDBs, if so required by the application.

Furthermore, by implementing the specialised integrity checking as a meta-interpreter, we are not stuck with one particular method: by adapting the meta-interpreter, we can implement different strategies depending on the application at hand.

However, to the best of our knowledge, the idea based on partially evaluating a meta-interpreter, was never actually implemented and in the second part of this paper we provide the first practical realisation.

A second line of motivation has grown from our previous attempts to promote partial deduction toward a broader community in software development research and industry, as a mature technology for automatic optimisation of software. Although partial deduction is by now a well-accepted and frequently applied optimisation technique within the logic programming community, very few reports on successfully optimised applications have appeared in the literature. This is quite in contrast with work on partial evaluation for functional – and even imperative – languages, where several "success stories", e.g. in the areas of ray tracing [70], scientific computing [32] and simulation and modelling [1,2,91], have been published.

As such, a second motivation for this work has been to report on a very successful application of partial deduction: our experiments illustrating how specialisation can significantly improve on the best general purpose integrity checking techniques known in the literature.

The work has been conducted along the following steps:

- First, we selected one of the most efficient integrity checking methods currently available: the one by Lloyd et al., presented in [63,61] (see also Section 2). We will refer to it as LST.
- We observed that this method itself is not very amenable to program specialisation. The reasons for this are discussed in Section 4 (see also Section 7). The main reason is that one of the functionalities in the LST method is to produce instantiated versions of the integrity checks. In a meta-interpreter implementation, this functionality can either be accomplished declaratively, using the ground representation, or, it can be implemented using non-declarative features in the non-ground representation. However, both these solutions put very high demands on a specialiser to be able to cope with them efficiently. Current specialisers are incapable of effectively optimising either of them.
- We therefore introduced a simplified integrity checking method, which does not produce instantiations of checks. This method is less efficient than LST, but (due to the absence of producing instantiations) it can be easily implemented as a meta-interpreter in the so called "mixed" representation: the ground representation, lifted to the non-ground one for resolution. Our hope and expectation were that the loss of efficiency resulting from not producing instantiated versions of checks would automatically be regained through specialisation. An expectation which turned out to be completely justified.
- On the side, we enhanced the resulting integrity checking method with a new, more refined feature, involving incremental checking (see Section 2). Although

this enhancement somewhat compensates for the precision loss of our interpreter with respect to LST (see also Section 6.1), the resulting technique is (in general) still considerably less efficient than LST itself (see Section 6.3). [4]

- We then applied a semi-off-line specialiser, developed in previous work [47,48], to the integrity checker with respect to different databases and update patterns. The results showed that the specialised integrity checks were extensively faster than an efficient implementation of the LST method.

The contributions of this work are on two different levels. On the lowest level:

- Through program specialisation, we developed specialised integrity checks which perform significantly better than the best general purpose integrity checking methods currently available. As such, the application of specialisation to this area has been shown to work exceptionally well.
- On the side, we also introduced an enhancement to the integrity checking method we started off from, consisting of an incremental checking behaviour. Nevertheless, the resulting method (without specialisation) is in general not competitive with LST.

On a higher level, our main contribution is on specialisation engineering. In the area of functional programming, it has long been generally accepted that effective program specialisation requires (re-)engineering of the input program and/or of its binding times. In logic programming, specialisation is still very much perceived as a "push button", fully automatic optimisation.

The main lesson we learned from this work is that reengineering of the input program may indeed be crucial for successful specialisation. More specifically:

- For some algorithms (e.g. LST style integrity checking) it seems unclear how they could be coded so that they become amenable to specialisation.
- In such cases, it may be beneficial to degrade the efficiency of the algorithm, in view of a following specialisation. Some arguments can be:
  - Introducing overhead (in our case aimed at pruning the search space by detecting irrelevant derivations) is acceptable, if it can be executed during specialisation.
  - Allowing redundant failing computations is acceptable, if they can be detected during specialisation.
  - Not producing the most instantiated versions of datastructures may be acceptable, if they can be produced during specialisation.
- In the more specific context of meta-interpreters, our experience is that, if possible, the "mixed" representation should be preferred over the fully ground one, and that annotations are useful, since automatic unfolding is still difficult.

Finally note that all this reengineering only needs to be performed once. In particular, in the case of our application, the reengineered (and annotated) integrity checker can be specialised over and over for different databases and different update patterns.

An earlier version of this article appeared in [53].

[4] We might in fact have clarified the contributions of this paper better by not introducing the incremental solving enhancement, so that the achievements of the specialisation would have become more obvious. But, as we discuss further on, this has been a specialisation engineering experiment and deliberately downgrading the results was not a choice we wanted to make.

2. Deductive databases and integrity checking

We assume the reader to be familiar with the standard notions of logic programming. Introductions to logic programming can be found in [3,59]. We use the convention to represent logical variables by uppercase letters like X, Y. Predicates and functors will be represented by lowercase letters like p, q, f, g. Finally, we allow SLDNF-derivations to be *incomplete*, i.e. neither leading to success nor failure, but to a goal where no literal has been selected for a further derivation step.

In this section we present some essential background in deductive databases and integrity checking. We also present a new method for specialised integrity checking, which we will later on implement as a meta-interpreter.

Definition 2.1. A *clause* is a first-order formula of the form *Head* $\leftarrow$ *Body* where *Head* is an atom and *Body* is a conjunction of literals. A *deductive database* is a set of clauses.

A *fact* is a clause with an empty body, while an *integrity constraint* is a clause of the form *false* $\leftarrow$ *Body*. A *rule* is a clause which is neither a fact nor an integrity constraint. As is well known, more general rules and constraints can be reduced to this format through the transformations proposed in [62]. Constraints in this format are referred to as inconsistency indicators in [84].

For the purposes of this paper, it is convenient to consider a database to be *inconsistent*, or violating the integrity constraints, iff *false* is derivable in the database via SLDNF-resolution. Other views of inconsistency exist and some discussions can for instance be found in [13]. Note that we do not require a deductive database to be range-restricted. This is because our notion of integrity is based on SLDNF-resolution, which always gives the same answer irrespective of the underlying language (see e.g. [85]). However, range-restriction is still useful as it ensures that no SLDNF-refutation will flounder.

As pointed out above, integrity constraints play a crucial role in several logic programming based research areas. It is however probably fair to say that they received most attention in the context of (relational and) deductive databases. Addressed topics are, among others, constraint satisfiability, semantic query optimisation, system supported or even fully automatic recovery after integrity violation and efficient constraint checking upon updates. It is the latter topic that we focus on in this paper.

Two seminal contributions, providing first treatments of efficient integrity constraint checking upon updates in a deductive database setting, are [21,63,61]. In essence, what is proposed is reasoning forwards from an explicit addition or deletion, computing indirectly caused implicit *potential updates*.

Consider the following clause:

$$p(X, Y) \leftarrow q(X), r(Y)$$

The addition of $q(a)$ might cause implicit additions of $p(a, Y)$-like facts. Which instances of $p(a, Y)$ will actually be derivable depends of course on r. Moreover, some or all such instances might already be provable in some other way. Propagating such potential updates through the program clauses, we might hit upon the possible addition of *false*. Each time this happens, a way in which the update might endanger integrity has been uncovered. It is then necessary to evaluate the (properly instanti-

ated) body of the affected integrity constraint to check whether *false* is actually provable in this way.

Propagation of potential updates, along the lines proposed in [63,61], can be formalised as follows.

Definition 2.2. A *database update*, U, is a triple $\langle Db^+, Db^=, Db^- \rangle$ of mutually disjoint deductive databases. We say that δ is an *SLDNF-derivation after* U for a goal G iff δ is an SLDNF-derivation for $Db^+ \cup Db^= \cup \{G\}$. Similarly δ is an *SLDNF-derivation before* U for G if δ is an SLDNF-derivation for $Db^- \cup Db^= \cup \{G\}$.

Db^- are the clauses removed by the update and Db^+ are the clauses which are added by the update. Thus, $Db^- \cup Db^=$ represents the database state before the update and $Db^+ \cup Db^=$ represents the database state after the update.

Below, $mgu^*(A,B)$ represents a particular idempotent and relevant [5] *mgu* of $\{A, B'\}$, where B' is obtained from B by renaming apart (w.r.t. A). If no such unifier exists then $mgu^*(A,B) = fail$. The operation mgu^* has the interesting property that $mgu^*(A,B) = fail$ iff A and B have no common instance (for a proof see e.g. [52]).

Definition 2.3. Given a database update $U = \langle Db^+, Db^=, Db^- \rangle$, we define the set of *positive potential updates* $pos(U)$ and the set of *negative potential updates* $neg(U)$ inductively as follows.

$$pos^0(U) = \{A \mid A \leftarrow Body \in Db^+\},$$

$$neg^0(U) = \{A \mid A \leftarrow Body \in Db^-\},$$

$$\begin{aligned} pos^{i+1}(U) = {} & \{A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, B, \ldots, \\ & C \in pos^i(U) \text{ and } mgu^*(B,C) = \theta\} \\ \cup \quad & \{A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, \neg B, \ldots, \\ & C \in neg^i(U) \text{ and } mgu^*(B,C) = \theta\}, \end{aligned}$$

$$\begin{aligned} neg^{i+1}(U) = {} & \{A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, B, \ldots, \\ & C \in neg^i(U) \text{ and } mgu^*(B,C) = \theta\} \\ \cup \quad & \{A\theta \mid A \leftarrow Body \in Db^=,\ Body = \ldots, \neg B, \ldots, \\ & C \in pos^i(U) \text{ and } mgu^*(B,C) = \theta\}, \end{aligned}$$

$$pos(U) = \bigcup_{i \geq 0} pos^i(U),$$

$$neg(U) = \bigcup_{i \geq 0} neg^i(U).$$

These sets can be computed through a bottom-up fixpoint iteration.

Example 2.4. Let $Db^+ = \{man(a) \leftarrow\}$, $Db^- = \emptyset$ and let the following clauses be the rules of $Db^=$:

[5] I.e. *all* its variables occur in $\{A, B'\}$.

$mother(X,Y) \leftarrow parent(X,Y),\ woman(X)$

$father(X,Y) \leftarrow parent(X,Y),\ man(X)$

$false \leftarrow man(X),\ woman(X)$

$false \leftarrow parent(X,Y),\ parent(Y,X)$

For the update $U = \langle Db^+, Db^=, Db^- \rangle$, we then obtain, independently of the facts in $Db^=$, that $pos(U) = \{man(a), father(a,\_), false\}$ and $neg(U) = \emptyset$.

Simplified integrity checking, along the lines of the Lloyd, Topor and Sonenberg (LST) method [63,61], then essentially boils down to evaluating the corresponding (instantiated through θ) *Body* every time a *false* fact gets inserted into some $pos^i(U)$. In Example 2.4, one would thus only have to check $\leftarrow man(a), woman(a)$. In practice, these tests can be collected, those that are instances of others removed, and the remaining ones evaluated in a separate constraint checking phase. The main difference with the exposition in [63,61] is that we calculate $pos(U)$ and $neg(U)$ in one step instead of in two. [6]

Note that in the above definition we do not test whether an atom $A \in pos(U)$ is a "real" update, i.e. whether A is actually derivable after the update (this is what is called the *phantomness* test) and whether A was indeed not derivable before the update (this is called the *idleness* test). A similar remark can be made about the atoms in $neg(U)$. Other proposals for simplified integrity checking often feature more precise (but more laborious) update propagation. The method by Decker [21], for example, performs the phantomness test and computes *induced updates* rather than just potential updates. Many other solutions are possible, all with their own weak as well as strong points. An overview of the work during the 1980s is offered in [13]. A clear exposition of the main issues in update propagation, emerging after a decade of research on this topic, can be found in [43]. Ref. [16] compares the efficiency of some major strategies on a range of examples. Finally, recent contributions can be found in, among others, [15,23,45,84,46].

Concentrating on potential updates has the advantage that one does not have to access the entire database for the update propagation. In fact, Definition 2.3 does not reference the facts in $Db^=$ at all (only clauses with at least one literal in the body are used). For a lot of examples, like databases with a large number of facts, this leads to very good efficiency (see, e.g., the experiments in [16,84]). It, however, also somewhat restricts the usefulness of a method based solely on Definition 2.3 when the rules and integrity constraints change more often than the facts.

Based upon Definition 2.3 we now formalise some simplification methods in more detail. The following definition uses the sets $pos(U)$ and $neg(U)$ to obtain more specific instances of goals and detect whether the proof tree of a goal is potentially affected by an update.

Definition 2.5. Given a database update U and a goal $G = \leftarrow L_1, \ldots, L_n$, we define:

[6] This approach should be more efficient, while yielding the same result, because in each iteration step the influence of an atom C is independent of the other atoms currently in pos^i and neg^i.

$$\Theta_U^+(G) = \{\theta \mid C \in pos(U),\ mgu^*(L_i, C) = \theta,\ L_i \text{ is a positive literal and } 1 \leqslant i \leqslant n\} \cup \{\theta \mid C \in neg(U),\ mgu^*(A_i, C) = \theta,\ L_i = \neg A_i \text{ and } 1 \leqslant i \leqslant n\},$$

$$\Theta_U^-(G) = \{\theta \mid C \in neg(U),\ mgu^*(L_i, C) = \theta,\ L_i \text{ is a positive literal and } 1 \leqslant i \leqslant n\} \cup \{\theta \mid C \in pos(U),\ mgu^*(A_i, C) = \theta,\ L_i = \neg A_i \text{ and } 1 \leqslant i \leqslant n\}.$$

We say that G is *potentially added by* U iff $\Theta_U^+(G) \neq \emptyset$. Also, G is *potentially deleted by* U iff $\Theta_U^-(G) \neq \emptyset$.

Note that trivially $\Theta_U^+(G) \neq \emptyset$ iff $\Theta_U^+(\leftarrow L_i) \neq \emptyset$ for some literal L_i of G. The LST-method in [61] can roughly be seen as calculating $\Theta_U^+(\leftarrow Body)$ for each body *Body* of an integrity constraint and then evaluating the simplified constraint $false \leftarrow Body\ \theta$ for every $\theta \in \Theta_U^+(\leftarrow Body)$.

For the Example 2.4 above we obtain:

$$\Theta_U^+(\leftarrow man(X),\ woman(X)) = \{\{X/a\}\} \quad \text{and}$$
$$\Theta_U^+(\leftarrow parent(X,Y),\ parent(Y,X)) = \emptyset$$

and thus obtain the following set of simplified integrity constraints:

$$\{false \leftarrow man(a),\ woman(a)\}.$$

Checking this simplified constraint is of course much more efficient than entirely re-checking the unsimplified integrity constraints of Example 2.4.

In the remainder of this section, we present a new integrity checking method, which uses the substitutions Θ_U^+ slightly differently. This method has the advantage that, as we will see later, it can be implemented in a manner more amenable to specialisation than the LST-method.

First though, we characterise derivations in a database after an update, which were not present before the update.

Definition 2.6. Let $U = \langle Db^+, Db^=, Db^- \rangle$ be a database update and let δ be an SLDNF-derivation after U for a goal G. A derivation step of δ will be called *incremental* iff it resolves a positive literal with a clause from Db^+ or if it selects a ground negative literal $\neg A$ such that $\leftarrow A$ is potentially deleted by U. We say that δ is *incremental* iff it contains at least one incremental derivation step.

The treatment of negative literals in the above definition is not optimal. In fact "$\leftarrow A$ is potentially deleted by U" does not guarantee that the same derivation does not exist in the database state prior to an update. However an optimal criterion, due to its complexity, has not been implemented in the current approach.

Lemma 2.7. *Let G be a goal and U a database update. If there exists an incremental SLDNF-derivation after U for G, then G is potentially added by U.*

Proof. Let $U = \langle Db^+, Db^=, Db^- \rangle$ and let δ be the incremental SLDNF-derivation after U for G. We define δ' to be the incremental SLDNF-derivation for G after U

obtained by stopping at the first incremental derivation step of δ. Let $G_0 = G, G_1, \ldots, G_k$, with $k > 0$, be the sequence of goals of δ'. We will now prove by induction on the length k of δ' that G_0 is potentially added by U.

Base Case ($k = 1$): This means that only one derivation step has been performed, which must therefore be incremental. There are two possibilities: either a positive literal $L_i = A_i$ or a negative literal $L_i = \neg A_i$ has been selected inside G_0. In the first case the goal G_0 has been resolved with a standardised apart [7] clause $A \leftarrow Body \in Db^+$ with $mgu(A_i, A) = \theta$. Thus by Definition 2.3 we have $A \in pos^0(U)$ and by Definition 2.5 we obtain $\theta \in \Theta_U^+(G_0)$. In the second case we must have $\Theta_U^-(\leftarrow A_i) \neq \emptyset$ and by Definition 2.5 there exists a $C \in neg(U)$ such that $mgu^*(A_i, C) = \theta$. Hence we know that $\theta \in \Theta_U^+(\leftarrow G_0)$. So, in both cases $\Theta_U^+(\leftarrow G_0) \neq \emptyset$, i.e. $G = G_0$ is potentially added by U.

Induction hypothesis: For $1 \leqslant k \leqslant n$ we have that G_0 is potentially added by U.

Induction step ($k = n + 1$): We can first apply the induction hypothesis on the incremental SLDNF-derivation for G_1 after U consisting of the last n steps of δ (i.e.whose sequence of goals is $G_1, \ldots, G_{n+1}$) to deduce that G_1 is potentially added by U.

Let $G_1 = \leftarrow L_1, \ldots, L_n$. We know that for at least one literal L_i we have that $\Theta_U^+(\leftarrow L_i) \neq \emptyset$.

If a negative literal has been selected in the derivation step from G_0 to G_1 then G_0 is also potentially added, because all the literals L_i also occur unchanged in G_0.

If a positive literal L'_j has been selected in the derivation step from G_0 to G_1 and resolved with the (standardised apart) clause $A \leftarrow B_1, \ldots, B_q \in Db^=$, with $mgu(L'_j, A) = \theta$, we have: $G_0 = \leftarrow L'_1, \ldots, L'_j, \ldots, L'_r, G_1 = \leftarrow (L'_1, \ldots, L'_{j-1}, B_1, \ldots, B_q, L'_{j+1}, \ldots, L'_r)\theta$.

There are again two cases. Either there exists a L'_p, with $1 \leqslant p \leqslant r \wedge p \neq j$, such that $\Theta_U^+(\leftarrow L'_p\theta) \neq \emptyset$. In that case we have for the more general goal $\leftarrow L'_p$ that $\Theta_U^+(\leftarrow L'_p) \neq \emptyset$ [8] and therefore G_0 is potentially added.

In the other case there must exist a B_p, with $1 \leqslant p \leqslant q$, such that $\Theta_U^+(\leftarrow B_p\theta) \neq \emptyset$. If B_p is a positive literal, we have by Definition 2.5 for some $C \in pos(U)$ that $mgu^*(B_p\theta, C) = \sigma$. Therefore, by Definition 2.3, we know that there is an element $A' \in pos(U)$ which is more general than $A\theta\sigma$. As $A\theta$ is an instance of L'_j, L'_j and A' have the common instance $A\theta\sigma$ and thus $mgu^*(L'_j, A')$ must exist and we can thus conclude that $\Theta_U^+(\leftarrow L'_j) \neq \emptyset$ and that $G = G_0$ is potentially added.

The proof is almost identical for the case that B_p is a negative literal. □

Definition 2.8. Let U be a database update, δ a (possibly incomplete) SLDNF-derivation after U for G_0 and $G_0, G_1, \ldots$ be the sequence of goals of δ. We say that δ is a *relevant SLDNF-derivation after* U for G_0 iff for each G_i we either have that G_i is

[7] So far we have not provided a formal definition of the notion of "standardising apart". Several ones, correct and incorrect, exist in the literature (see e.g. the discussion in [40] or [25]). Just suppose for the remainder of this proof that fresh variables, not occurring "anywhere else", are used.

[8] Note that this is *not* the case if we use just the *mgu* without standardising apart inside Definition 2.5. This technical detail has been overlooked in [63,61]. Take for instance $L'_p = p(X)$ and $\theta = \{X/f(Y)\}$. Then $L'_p\theta$ unifies with $p(f(X)) \in pos(U)$ and the more general L'_p does not!

potentially added by U or δ_i is incremental after U, where δ_i is the sub-derivation leading from G_0 to G_i.

A refutation being a particular derivation we can specialise the concept and define *relevant refutations*. The following theorem will form the basis of our method for performing specialised integrity checking.

Theorem 2.9 (*Incremental integrity checking*). *Let $U = \langle Db^+, Db^=, Db^- \rangle$ be a database update such that there is no SLDNF-refutation before U for the goal $\leftarrow$ false. Then $\leftarrow$ false has an SLDNF-refutation after U iff $\leftarrow$ false has a relevant refutation after U.*

Proof. ($\Leftarrow$): If $\leftarrow$ *false* has a relevant refutation then it trivially has a refutation, namely the relevant one.

($\Rightarrow$): The refutation must be incremental, because otherwise the derivation is also valid for $Db^= \cup Db^-$ and we have a contradiction. We can thus use Lemma 2.7 to infer that the derivation conforms to Definition 2.8 and is relevant. □

In other words, if we know that the integrity constraints of a deductive database were not violated before an update, then we only have to search for a *relevant* refutation of $\leftarrow$ *false* in order to check the integrity constraints after the update. Observe that, by definition, once a derivation is not relevant, it cannot be extended into a relevant one.

The method can be illustrated by re-examining Example 2.4. The goals in the SLD-tree in Fig. 1 are annotated with their corresponding sets of substitutions Θ_U^+. The SLD-derivation leading to $\leftarrow$ *parent*(X, Y), *parent*(Y, X) is not relevant and can therefore be pruned. Similarly all derivations descending from the goal $\leftarrow$ *man*(X), *woman*(X) which do not use $Db^+ = \{man(a) \leftarrow\}$ are not relevant either and can also be pruned. However the derivation leading to $\leftarrow$ *woman*(a) is incremental and is relevant even though $\leftarrow$ *woman*(a) is not potentially added.

The above method can be seen as a refinement of the LST method [61], in the sense that Θ_U^+ is not only used to simplify the integrity constraints at the topmost level (i.e. affecting the bodies of integrity constraints), but is used throughout the testing of the integrity constraints to prune non-relevant branches. An example where this aspect is important will be presented in Section 6.

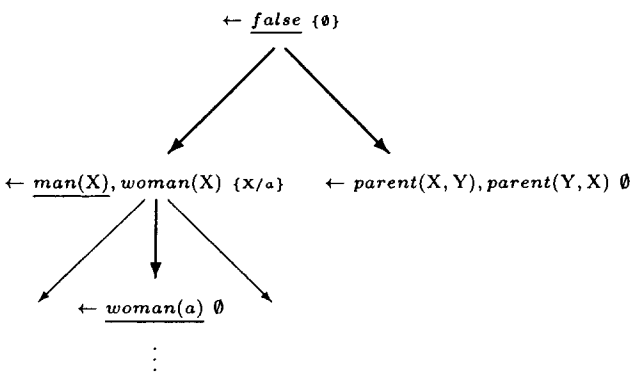

Fig. 1. SLDNF-tree for Example 2.4.

However, the LST method [61] not only removes integrity constraints but also *instantiates* them, possibly generating several specialised integrity constraints for a single unspecialised one. This instantiation, although not guaranteed to improve the efficiency, often considerably reduces the number of matching facts. This aspect is often vital in practice for improving the efficiency of the integrity checks. The Definition 2.8 of relevant derivations does not use Θ_U^+ to instantiate intermediate goals. This means that, although the pruning is more refined, [9] the method used on its own has probably little practical value for larger databases. [10] However, the absence of instantiations allows a rather simple implementation of the method as a meta-interpreter, amenable to effective partial evaluation. As it will turn out, the missing instantiations will then often be performed by partial evaluation. Furthermore, some of the pruning will have already taken place during partial evaluation, resulting in little or no overhead due to the extra pruning complexity. The results in Section 6 will confirm this.

3. Integrity checking by a meta-interpreter

3.1. Specialised update procedures

Specialised integrity checking can be implemented through a meta-interpreter, manipulating updates and databases as object level expressions. As we already mentioned, a major benefit of such a meta-programming approach lies in the flexibility it offers: Any particular propagation and simplification strategy can be incorporated into the meta-program.

Furthermore, by partial evaluation of this meta-interpreter, we may (in principle) be able to pre-compile the integrity checking for certain update patterns. Let us re-examine Example 2.4. For the *concrete update* of Example 2.4, with $Db^+ = \{man(a) \leftarrow\}$, a meta-interpreter implementing the method of the previous section would try to find a refutation for $\leftarrow false$ in the manner outlined in Fig. 1. By specialising this meta-interpreter for an *update pattern* $Db^+ = \{man(\mathscr{A}) \leftarrow\}$, $Db^- = \emptyset$, where $\mathscr{A}$ is not yet known, one might (hopefully) obtain a *specialised update procedure*, efficiently checking integrity, essentially as follows:

$$inconsistent(add(man(A))) \leftarrow evaluate(woman(A))$$

Given the *concrete* value for $\mathscr{A}$, this procedure will basically check consistency in a similar manner to the unspecialised meta-interpreter, but will do this much more efficiently, because the propagation, simplification and evaluation process is already *pre-compiled.* For instance, the derivation in Fig. 1 leading to $\leftarrow parent(X,Y)$, $parent(Y,X)$ has already been pruned at specialisation time. Similarly all derivations descending from the goal $\leftarrow man(X)$, $woman(X)$, which do not use Db^+, have also already been pruned at specialisation time. Finally, the specialised update procedure no longer has to calculate $pos(U)$ and $neg(U)$ for the concrete update U. All of this

[9] But which will also result in some overhead when evaluating the integrity constraints.

[10] Although one could actually easily adapt Definition 2.8 to use Θ_U^+ for instantiating goals and Theorem 2.9 would still be valid.

can lead to very high efficiency gains. Furthermore, for the integrity checking method we have presented in the previous section, the same specialised update procedure can be used as long as the rules and integrity constraints do not change (i.e. Db^+ and Db^- only contain facts). For example, after any concrete update which is an instance of the update pattern above, the specialised update procedure remains valid and does not have to be re-generated.

Refs. [90,84] both explicitly address this compilation aspect. Their approaches are, however, more limited in some important respects and both use ad hoc techniques and terminology instead of well-established and general apparatus provided by meta-interpreters and partial deduction (the main concern of [84] is to show why using inconsistency indicators instead of integrity constraints is relevant for efficiency and a good idea in general).

In this section we will present a meta-interpreter for specialised integrity checking which is based on Theorem 2.9. This meta-interpreter will act on object level expressions (terms, atoms, goals, clauses,...) which represent the deductive database under consideration. So, before presenting the meta-interpreter in more detail, it is advisable to discuss the issue of representing these object level expressions at the meta-level, i.e. inside the meta-interpreter.

3.2. The ground, non-ground and mixed representations

In logic programming, there are basically two different approaches to representing an object level expression, say the atom $p(X,a)$, at the meta-level. In the first approach one uses the term $p(X,a)$ as the object level representation. This is called a *non-ground* representation, because it represents an object level variable by a meta-level variable. In the second approach one would use something like the term $struct(p,[var(1),struct(a,[\,])])$ to represent the object level atom $p(X,a)$. This is called a *ground* representation, as it represents an object level variable by a ground term. Fig. 2 contains some further examples of the particular ground representation which we will use throughout this paper. From now on, we use "$\mathscr{T}$" to denote the ground representation of a term $\mathscr{T}$.

The ground representation has the advantage that it can be treated in a *purely declarative* manner, while for many applications the non-ground representation requires the use of extra-logical built-in's (like `var/1` or `copy/2`). The non-ground representation also has semantical problems (although they were solved to some extent in [20,66,67]). The main advantage of the non-ground representation is that the meta-program can use the *underlying unification* mechanism, while for the ground representation an explicit unification algorithm is required. This (currently) induces a difference in speed reaching several orders of magnitude and can pose some subtle problems for specialisation [56,51]. The current consensus in the logic programming

Object level	Ground representation
X	$var(1)$
c	$struct(c,[])$
$f(X,a)$	$struct(f,[var(1),struct(a,[])])$
$p \leftarrow q$	$struct(clause,[struct(p,[]),struct(q,[])])$

Fig. 2. A ground representation.

community is that both representations have their merits and the actual choice depends on the particular application. For a more detailed discussion we refer the reader to [35,36,8,51], the conclusion of [66] or the extended version of [56].

Sometimes however, it is possible to combine both approaches. This was first exemplified by Gallagher in [29,30], where a (declarative) meta-interpreter for the ground representation is presented. From an operational point of view, this meta-interpreter lifts the ground representation to the non-ground one for resolution (an alternate declarative view is discussed below). We will call this approach the *mixed* representation, as object level goals are in non-ground form while the object programs are in ground form. [11] With that technique we can use the versatility of the ground representation for representing object level programs (but not goals), while still remaining reasonably efficient. Furthermore, as demonstrated by Gallagher in [29] and by the experiments in this paper, partial evaluation can in this way sometimes completely remove the overhead of the ground representation. Performing a similar feat on a meta-interpreter using the full ground representation with explicit unification is much harder and has, to the best of our knowledge, not been accomplished yet (for some promising attempts see [34,33,9] or [56]).

Instead of using the mixed representation, one might also ignore the semantical problems and think of simply using the non-ground representation to implement our meta-interpreter. There are actually several reasons why we have chosen not to do so:

- One disadvantage of the non-ground representation is that it is more difficult to *specify partial knowledge* for partial evaluation. Suppose that we know that a given atom (for instance the head of a fact that will be added to a deductive database) will be of the form $man(\mathscr{T})$, where $\mathscr{T}$ is an as of yet unknown constant. In the ground representation this knowledge can be expressed as $struct(man, [struct\ (C, [\,])])$. However, in the non-ground representation we have to write this as $man(X)$, which is unfortunately less precise, as the variable X now no longer represents only constants but stands for any term. [12]
- There is unfortunately another caveat to using the non-ground representation: either the object program has to be stored explicitly using meta-program clauses – instead of using a term-representation of the object program – or non-logical built-ins like *copy*/2 have to be used to perform the standardising apart. Fig. 3 illustrates these two possibilities. Note that without the *copy* in Fig. 3 the second meta-interpreter would incorrectly fail for the given query. For our application this means that, on the one hand, using the non-logical copying approach unduly complicates the specialisation task while at the same time leading to a serious efficiency bottleneck. On the other hand, using the clause representation, implies that representing updates to a database becomes much more cumbersome. Basically we also have to encode the updates explicitly as meta-program clauses, thereby making dynamic meta-programming (see e.g. [35]) impossible.

[11] A similar technique was used in the self-applicable partial evaluator LOGIMIX [71,37]. Hill and Gallagher [35] also provide a recent account of this style of writing meta-interpreters.

[12] A possible way out is to use the $=../2$ built-in and represent the atom by $man(X)$, $X = ..[C]$. This requires that the partial evaluator provides non-trivial support for the built-in $=../2$ (to ensure for instance that the information about X, provided by $X = ..[C]$, is properly used and propagated).

1. Using a Clause Representation	*2. Using a Term Representation*
$solve([]) \leftarrow$ $solve([H\|T]) \leftarrow$ $\quad clause(H, B)$ $\quad solve(B), solve(T)$ $clause(p(X), []) \leftarrow$	$solve(P, []) \leftarrow$ $solve(P, [H\|T]) \leftarrow$ $\quad member(Cl, P), copy(Cl, cl(H, B))$ $\quad solve(P, B), solve(P, T)$
$\leftarrow solve([p(a), p(b)])$	$\leftarrow solve([cl(p(X), [])], [p(a), p(b)])$

Fig. 3. Two non-ground meta-interpreters with $\{p(X) \leftarrow\}$ as object program.

In summary, the most promising option for our application is to use the mixed representation.

As such, our meta-interpreter will contain a predicate, called *make_non_ground*, which "lifts" a ground term to a non-ground one. For instance, the query

$$\leftarrow make\_non\_ground(struct(f, [var(1), var(2), var(1)]), X)$$

succeeds with a computed answer similar to

$$\{X/struct(f, [\_49, \_57, \_49])\}.$$

The variables _49 and _57 are fresh variables (whose actual names may vary and are not important). The code for this predicate is presented in Fig. 4 and a simple meta-interpreter based on it can be found in Fig. 5. This code is a variation of the *Instance-Demo* meta-interpreter in [35], where the predicate *make_non_ground*/2 is called *InstanceOf12*. Indeed, although operationally *make_non_ground*/2 lifts a ground term to a non-ground term, declaratively (and when typing the arguments) the second argument of *make_non_ground*/2 can be seen as representing all ground terms which are instances of the first argument; hence the names *InstanceOf12* and *Instance-Demo*. Also note that, in contrast to the original meta-interpreter presented in [29], these meta-interpreters can be executed without specialisation. One can even execute

$$\leftarrow make\_non\_ground(struct(man, [struct(C, [\ ])]), X)$$

and obtain the computed answer $\{X/struct(man, [struct(C, [\ ])])\}$. (However, the goal $\leftarrow make\_non\_ground(struct(man, [C]), X)$ has an infinite number of computed an-

```
make_non_ground(GrTerm, NgTerm) ←
    mng(GrTerm, NgTerm, [], _Sub)

mng(var(N), X, [], [sub(N, X)]) ←
mng(var(N), X, [sub(M, Y)|T], [sub(M, Y)|T1]) ←
    (N = M → (T1 = T, X = Y) ; mng(var(N), X, T, T1))
mng(struct(F, GrArgs), struct(F, NgArgs), InSub, OutSub) ←
    l_mng(GrArgs, NgArgs, InSub, OutSub)

l_mng([], [], Sub, Sub) ←
l_mng([GrH|GrT], [NgH|NgT], InSub, OutSub) ←
    mng(GrH, NgH, InSub, InSub1),
    l_mng(GrT, NgT, InSub1, OutSub)
```

Fig. 4. Lifting the ground representation.

```
solve(Prog,[]) ←
solve(Prog,[H|T]) ←
    non_ground_member(struct(clause,[H|Body]),Prog),
    solve(Prog,Body),
    solve(Prog,T)

non_ground_member(NonGrTerm,[GrH|GrT]) ←
    make_non_ground(GrH,NonGrTerm)
non_ground_member(NonGrTerm,[GrH|GrT]) ←
    non_ground_member(NonGrTerm,GrT)
```

Fig. 5. An interpreter for the ground representation.

swers.) Observe that in Fig. 4 we use an if-then-else construct, written as (`if` → `then; else`), which for the time being we assume to be declarative (even the Prolog if-then-else will behave in a declarative way, because the first argument to *make_non_ground* will always be ground).

We can now use Theorem 2.9 to extend the interpreter in Fig. 5 for specialised integrity checking. Based on Theorem 2.9, we know that we can stop resolving a goal G when it is not potentially added, unless we have performed an *incremental* resolution step earlier in the derivation.

The skeleton of our meta-interpreter in Fig. 6 implements this idea (the full Prolog code can be found in Appendix A). The argument Updt contains the ground representation of the update $\langle Db^+, Db^=, Db^- \rangle$.

The predicate *resolve_incrementally/3* performs incremental resolution steps (according to Definition 2.6) and *resolve_unincrementally/3* performs non-incremental ones. The predicate *potentially_added/2* tests whether a goal is potentially added by an update based on Definition 2.5. Specialised integrity checking now consists in calling

← *incremental_solve* ("$\langle Db^+,\ Db^=,\ Db^- \rangle$", ← *false*)

The query will succeed if the integrity of the database has been violated by the update.

```
incremental_solve(Updt,Goal) ←
  potentially_added(Updt,Goal),
  resolve(Updt,Goal)

resolve(Updt,Goal) ←
  resolve_unincrementally(Updt,Goal,NewGoal),
  incremental_solve(Updt,NewGoal)
resolve(Updt,Goal) ←
  resolve_incrementally(Updt,Goal,NewGoal),
  Updt ="⟨Db+,Db=,Db-⟩",
  solve("Db= ∪ Db+",NewGoal)
```

Fig. 6. Skeleton of the integrity checker.

4. Implementing specialised integrity checking

In this section study the implementation of the predicate *potentially_added/2*. Along the way, we also discuss why a full implementation of the LST method is (currently) not amenable to specialisation.

The rules of Definition 2.3 – at the basis of both *potentially_added/2* and the LST method – can be directly transformed into a simple logic program calculating the elements of $pos(U)$ and $neg(U)$. Making abstraction of the particular representation of clauses and programs, we might write *pos* like this:

$$
\begin{aligned}
&pos(\langle DB^+, DB^=, DB^- \rangle, A) \leftarrow \\
&\quad A \leftarrow \ldots \in DB^+ \\
&pos(\langle DB^+, DB^=, DB^- \rangle, A) \leftarrow \\
&\quad A \leftarrow \ldots, B, \ldots \in DB^=, \\
&\quad pos(\langle DB^+, DB^=, DB^- \rangle, B) \\
&pos(\langle DB^+, DB^=, DB^- \rangle, A) \leftarrow \\
&\quad A \leftarrow \ldots, \neg B, \ldots \in DB^=, \\
&\quad neg(\langle DB^+, DB^=, DB^- \rangle, B)
\end{aligned}
$$

Such a (naive) top-down implementation terminates for hierarchical databases and is quite easy to partially evaluate. It will, however, lead to a predicate which has multiple, possibly identical and/or subsumed [13] solutions. [14]

Also, in the context of the non-ground or the mixed representation, this predicate will instantiate the (non-ground) expressions under consideration. For instance if we add e.g. the clause

$$
\begin{aligned}
&potentially\_added(\langle DB^+, DB^=, DB^- \rangle, G) \leftarrow \\
&\quad G =\leftarrow \ldots, A, \ldots, \\
&\quad pos(\langle DB^+, DB^=, DB^- \rangle, A)
\end{aligned}
$$

then the goal G will be instantiated during execution. This means that, to ensure completeness, we would either have to backtrack and try out a lot of useless instantiations, [15] or collect all solutions and perform expensive subsumption tests to keep only the most general ones. The latter can only be done declaratively in the ground representation [55,56], the use of which we wanted to avoid (cf. Section 3.2). In the non-ground or mixed representation extra-logical primitives like `findall` (to collect all possible instantiations) or `numbervars` (for the subsumption tests) are required, both of which seem very hard to partially evaluate satisfactorily; e.g. effective partial evaluation of `findall` has to the best of our knowledge not been accomplished yet.

---

[13] A computed answer θ of a goal G is called *subsumed* if there exists another computed answer θ' of G such that $G\theta$ is an instance of $G\theta'$.

[14] Unless this program is written in a tabled logic programming language like XSB. However, specialisation of tabled logic programs is far from obvious, as e.g. even determinate unfolding can ruin termination [58]. Exploiting this alternative approach is therefore a topic for further research.

[15] It would also mean that we would have to extend Theorem 2.9 to allow for instantiation, but this is not a major problem.

Example 4.1. Let the following clauses be the rules of $Db^{=}$:

$$mother(X,Y) \leftarrow parent(X,Y),\ woman(X)$$

$$father(X,Y) \leftarrow parent(X,Y),\ man(X)$$

$$false \leftarrow mother(X,Y),\ father(X,Z)$$

Let $Db^{-} = \emptyset$, $Db^{+} = \{parent(a,b) \leftarrow, man(a) \leftarrow\}$ and $U = \langle Db^{+}, Db^{=}, Db^{-}\rangle$. A naive top-down implementation will succeed 3 times for the query

$$\leftarrow potentially\_added(\text{“}U\text{”},\ false)$$

and twice for the query

$$\leftarrow potentially\_added(\text{“}U\text{”},\ father(X,Y))$$

with computed answers $\{X/a\}$ and $\{X/a, Y/b\}$. Note that the solution $\{X/a, Y/b\}$ is "subsumed" by $\{X/a\}$ (which means that, if floundering is not possible, it is useless to instantiate the query by applying $\{X/a, Y/b\}$).

To summarise, if we want to include *instantiation* of goals in our integrity checker we can

- either use a simple implementation *without subsumption* tests, but which will perform a lot of redundant checking and is hopelessly inefficient (even after specialisation),
- or *perform subsumption* tests, in turn leading to
  - either the use of the ground representation and the currently associated problems in terms of efficiency and specialisation [55,56],
  - or the use of non-declarative features which are difficult to partially evaluate and will probably result in little specialisation given the existing techniques (cf. the end of Section 6.3 where only 10% speedup is achieved using an existing partial evaluator for full Prolog).

An implementation of an integrity checking method which relies on such instantiations, like, e.g., the LST method, would thus either be hopelessly inefficient or not amenable to effective specialisation using current techniques (although progress has been made e.g. in [54–56,51,58]). Also, in the context of fully *recursive* databases, a subsumption test (or something even more powerful) will be required to perform loop checking inside the integrity checker. Again, such an integrity checker will not be amenable to effective specialisation using current techniques.

It thus seems currently unclear how an implementation of the full LST method could be coded such that it becomes effectively amenable to specialisation. However, in the context of hierarchical databases and our new integrity checking method, we can solve this problem by wrapping calls to *potentially_added/2* into a *verify*(.) primitive, where *verify*(G) succeeds *once* with the *empty computed answer* if the goal G succeeds in any way and fails otherwise. This will solve the problem of duplicate and subsumed solutions. For instance, for Example 4.1 above, both

$$\leftarrow verify(potentially\_added(\text{“}U\text{”},\ false))$$

$$\leftarrow verify(potentially\_added(\text{“}U\text{”},\ father(X,Y)))$$

will succeed just once with the empty computed answer and no backtracking is required, as no instantiations are made.

The disadvantage of using *verify* is of course that now no instantiations at all are performed (which in general cut down the search space dramatically). However, as will see later, these instantiations can often be performed by program specialisation.

Unfortunately, for non-ground goals, the *verify*(.) primitive is not declarative. It can, however, be implemented with the Prolog if-then-else construct, whose semantics is still reasonably simple. Indeed, *verify*(*Goal*) can be translated into

((Goal → fail; true) → fail; true).

Also, as we will see in the next section, specialisation of the if-then-else poses much less problems than for instance the full blown cut. This was already suggested by O'Keefe in [74] and carried out by Takeuchi and Furukawa in [89]. So, given the current speed penalty of the ground representation [9], employing the if-then-else seems like a reasonable choice. In the next section we will first present a subset of full Prolog, called If-Then-Else-Prolog or just ITE-Prolog, and discuss how ITE-Prolog-programs can be specialised. ITE-Prolog of course contains the if-then-else, but includes several built-in's as well. Indeed, once the if-then-else is added, there are no additional semantical difficulties in handling some simple built-in's like `var/1`, `nonvar/1` and `=../2` (but not built-in's like `call/1` or `assert/1` which manipulate clauses and goals).

5. Partial evaluation of ITE-Prolog

5.1. Definition of ITE-Prolog

To define the syntax of ITE-Prolog-programs we partition the predicate symbols into two disjoint sets Π_{bi} (the predicate symbols to be used for built-in's) and Π_{cl} (the predicate symbols for user-defined predicates). A *normal atom* is then an atom which is constructed using a predicate symbol $\in \Pi_{cl}$. Similarly a *built-in atom* is constructed using a predicate symbol $\in \Pi_{bi}$. An *ITE-Prolog-atom* is either a normal atom, a built-in atom or it is an expression of the form $(if \rightarrow then; else)$ where $if, then$ and *else* are conjunctions of ITE-Prolog-atoms. An *ITE-Prolog-clause* is an expression of the form *Head* ← *Body* where *Head* is a normal atom and *Body* is a conjunction of ITE-Prolog-atoms.

Operationally, the if-then-else of ITE-Prolog behaves like the corresponding construct in Prolog [88,73]. The following informal Prolog clauses can be used to define the if-then-else [80]:

(If → Then; Else) :- If, !, Then.

(If → Then; Else) :- Else.

In other words, when the test If succeeds (for the first time) a local cut is executed and execution proceeds with the then part. Most uses of the cut can actually be mapped to if-then-else constructs [74].

Because the if-then-else contains a local cut, its behaviour is sensitive to the sequence of computed answers of the test-part. This means that the computation rule and the search rule have to be fixed in order to give a clear meaning to the if-then-else. From now on we will presuppose the Prolog left-to-right computation rule and the lexical search rule. SLD-trees and derivations following this convention will be called LD-trees and derivations.

The two ITE-Prolog-programs hereafter illustrate this point.

Program P_1	*Program* P_2
$q(X) \leftarrow (p(X) \rightarrow X = c;\ fail)$	$q(X) \leftarrow (p(X) \rightarrow X = c;\ fail)$
$p(a) \leftarrow$	$p(c) \leftarrow$
$p(c) \leftarrow$	$p(a) \leftarrow$

Using the Prolog computation and search rules, the query $\leftarrow q(X)$ will fail for program P_1, whereas it will succeed for P_2. All we have done is change the order of the computed answers for the predicate $p/1$. This implies that a partial evaluator for ITE-Prolog has to ensure the preservation of the *sequence* of computed answers. This for instance is not guaranteed by the partial deduction framework of [60], which only preserves the computed answers but not their sequence.

However, the semantics of the if-then-else remains reasonably simple and a straightforward denotational semantics [38], in the style of [4,79], can be given to ITE-Prolog. For example, suppose that we associate to each ITE-Prolog-literal L a denotation $|L|_P$ in the program P, which is a possibly infinite sequence of computed answer substitutions with an optional element $\perp$ at the end of (a finite sequence) to denote looping. Some examples are then $|true|_P = \langle\emptyset\rangle$, $|X = a|_P = \langle\{X/a\}\rangle$ and $|fail|_P = \langle\ \rangle$. In such a setting the denotation of an if-then-else construct can simply be defined by:

$$|(A \rightarrow B; C)|_P = \begin{cases} |B\theta_1|_P & \text{if } |A|_P = \langle\theta_1, \ldots\rangle, \\ |C|_P & \text{if } |A|_P = \langle\ \rangle, \\ \langle\perp\rangle & \text{if } |A|_P = \langle\perp\rangle. \end{cases}$$

5.2. *Specialising ITE-Prolog*

In order to preserve the sequence of computed answers, we have to address a problem related to the *left-propagation* of bindings. For instance, unfolding a non-leftmost atom in a clause might instantiate the atoms to the left of it or the head of the clause. In the context of extra-logical built-ins this can change the program's behaviour. But even without built-ins, this left-propagation of bindings can change the order of solutions which, as we have seen above, can lead to incorrect transformations for programs containing the if-then-else. In the example below, P_4 is obtained from P_3 by unfolding the non-leftmost atom $q(Y)$, thereby changing the sequence of computed answers.

Program P_3	*Program* P_4
$p(X,Y) \leftarrow q(X), \underline{q(Y)}$	$p(X,a) \leftarrow q(X)$
$q(a) \leftarrow$	$p(X,b) \leftarrow q(X)$
$q(b) \leftarrow$	$q(a) \leftarrow$
	$q(b) \leftarrow$
Sequence of computed answers for $\leftarrow p(X,Y)$	
$\langle p(a,a), p(a,b), p(b,a), p(b,b)\rangle$	$\langle p(a,a), p(b,a), p(a,b), p(b,b)\rangle$

This problem of left-propagation of bindings has been solved in various ways in the partial evaluation literature [77,76,83,82], as well as overlooked in some contributions (e.g. [27]). In the context of unfold/fold transformations of pure logic programs, preservation of the order of solutions, as well as left-termination, is handled, e.g., in [79,7].

In the remainder, we will use the techniques we described in [48] to specialise ITE-Prolog-programs. The method of [48] tries to strictly enforce the Prolog left-to-right selection rule. However, sometimes one does not want to select the leftmost atom, for instance because it is a built-in which is not sufficiently instantiated, or simply to ensure termination of the partial evaluation process. To cope with this problem, [48] extends the concept of LD-derivations and LD-trees to LDR-derivations and LDR-trees, which in addition to left-most resolution steps also contain *residualisation* steps. The latter remove the left-most atom from the goal and hide it from the left-propagations of bindings. Generating the residual code from LDR-trees is discussed in [48].

Unfolding inside the if-then-else is also handled in a rather straightforward manner. This is in big contrast to programs which contain the full blown cut. The reason is that the full cut can have an effect on all subsequent clauses defining the predicate under consideration. By unfolding, the scope of a cut can be changed, thereby altering its effect. The treatment of cuts therefore requires some rather involved techniques (see [14], [83,82] or [77,76]). The cut inside the if-then-else, however, is local and does not affect the reachability and meaning of other clauses. It is therefore much easier to handle by a partial evaluator. The following example illustrates this. Unfolding $q(X)$ in the program P_5 with the if-then-else poses no problems and leads to a correct specialisation. However, unfolding the same atom in the program P_6 written with the cut leads to an incorrect specialised program in which, e.g., $p(a)$ is no longer a consequence.

Program P_5	*Program* P_6
$p(X) \leftarrow (\underline{q(X)} \rightarrow fail; true)$	$p(X) \leftarrow \underline{q(X)}, !, fail$
	$p(X) \leftarrow$
$q(X) \leftarrow (X = a \rightarrow fail; true)$	$q(X) \leftarrow X = a, !, fail$
	$q(X) \leftarrow$
Unfolded Programs	
$p(X) \leftarrow ((X = a \rightarrow fail; true)$	$p(X) \leftarrow X = a, !, fail, !, fail$
$\rightarrow fail$	$p(X) \leftarrow !, fail$
$; true)$	$p(X) \leftarrow$

The exact details on how to unfold the if-then-else can be found in [48], where a freeness and sharing analysis are used to produce more efficient specialised programs by removing useless bindings as well as useless if-then-else tests. The latter often occur when predicates with output arguments are used inside the test-part of an if-then-else. A further improvement lies in generating multiple specialisations of a predicate call according to varying freeness information of the arguments.

In summary, partial evaluation of ITE-Prolog, can be situated somewhere between partial deduction of *pure* logic programs (see [60,31,10,69,68] and systems like

SP [30,29], SAGE [34,33] or more recently ECCE [50,51,57]) and partial evaluation of *full* Prolog (e.g. MIXTUS [83,82] or PADDY [76–78]).

5.3. *Some aspects of* LEUPEL

The partial evaluation system LEUPEL, includes all the techniques sketched so far in this section. The implementation originally grew out of [47]. In Section 6 we will apply this system to obtain specialised update procedures. Below we present two more aspects of that system, relevant for our application.

5.3.1. *Safe left-propagation of bindings*

At the end of Section 4 we pointed out that a disadvantage of using *verify* is that no instantiations at all are performed. Fortunately, a lot of these instantiations can be performed by the partial evaluation method, through pruning and safe left-propagation of bindings. Take for instance a look at the specialised update procedure in Fig. 9 (to be presented later in Section 6) and generated for the update $Db^+ = \{man(a) \leftarrow\}$, $Db^- = \emptyset$. This update procedure tests directly whether $woman(\mathscr{A})$ is a fact, whereas the original meta-interpreter of Fig. 6 would test whether there are facts matching $woman(X)$ and only afterwards prune all irrelevant branches. This instantiation performed by the partial evaluator is in fact the reason for the extremely high speedup figures presented in the results of Section 6. In a sense, part of the specialised integrity checking is performed by the meta-interpreter and part is performed by the partial evaluator (the instantiations).

The above optimisation was obtained by unfolding and pruning all irrelevant branches. In some cases we can also improve the specialised update procedures by performing a *safe* left-propagation of bindings. As we have seen in the previous subsection, left-propagation of bindings is in general unsafe in the context of ITE-Prolog (left-propagation is of course safe for purely declarative programs and is performed by ordinary partial deduction). There are, however, some circumstances where bindings can be left-propagated without affecting the correctness of the specialised program. The following example illustrates such a safe left-propagation, as well as its benefits for efficiency.

Example 5.1. Take the following clause, which might be part of a specialised update procedure.

$$incremental\_solve\_1(A) \leftarrow parent(X, Y),$$
$$(a\_test \rightarrow X = A, Y = b;\ X = A,\ Y = c)$$

Suppose that the computed answers of $parent/2$ are always grounding substitutions. This is always guaranteed for range restricted (see e.g. [13]) database predicates. In that case the binding $X = A$ can be left-propagated in the following way:

$$incremental\_solve\_1(A) \leftarrow \underline{X = A}, parent(X, Y),$$
$$(a\_test \rightarrow Y = b; Y = c)$$

This clause will generate the same sequence of computed answers than the original clause, but will do so much more efficiently. Usually, there will be lots of $parent/2$ facts and $incremental\_solve\_1$ will be called with A instantiated. Therefore, the second

clause will be much more efficient – the call to *parent* will just succeed once for every child of A instead of succeeding for the entire *parent* relation.

The LEUPEL partial evaluator contains a post-processing phase which performs conservative but safe left-propagation of bindings, common to *all* alternatives (like $X = A$ above). This guarantees that no additional choice points are generated but in the context of large databases this is usually not optimal. For instance, for the Example 5.1 above, we might also left-propagate the non-common bindings concerning X, thereby producing the following clauses:

$$incremental\_solve\_1(A) \leftarrow \underline{X = A, Y = b},\ parent(X, Y),$$
$$(a\_test \rightarrow true;\ fail)$$
$$incremental\_solve\_1(A) \leftarrow \underline{X = A, Y = c}, parent(X, Y),$$
$$(a\_test \rightarrow fail;\ true)$$

In general this specialised update procedure will be even more efficient. This indicates that the results in Section 6 can be even further improved.

5.3.2. Control of unfolding

The partial evaluator LEUPEL is a semi-offline (in the sense that part of the unfolding decisions are made off-line, see e.g. [37,17,86]) system, decomposed into three phases:

- the *annotation phase*, which annotates the program to be specialised,
- the *specialisation phase*, which performs the unfolding guided by the annotations of the first phase,
- the *post-processing phase*, which performs optimisation on the generated partial deductions (i.e. removes useless bindings, performs safe left-propagation of bindings, simplifies the residual code) and generates the residual program.

The annotation phase of LEUPEL is not yet automatic and must be performed by hand. On the positive side, this gives the knowledgeable user very precise control over the unfolding. [16]

Automatically unfolding a meta-interpreter in a satisfactory way is a non-trivial issue and has been the topic of a lot of contributions [44,87,75,72,6,33,65,64]. For the fully general case, this problem has not been solved yet. However, as we are in the context of hierarchical databases and as we can hand annotate the meta-interpreter, a fully satisfactory unfolding can be achieved.

Furthermore, given proper care, the same annotated meta-program can be used for *any* kind of update pattern. Therefore, investing time in annotating the meta-interpreter of Appendix A – which only has to be done *once* – gives high benefits for all consecutive applications and the second and third phases of LEUPEL will then be able to derive specialised update procedures *fully automatically*, as exemplified by the prototype [49].

[16] Especially since some quite refined annotations are provided for. For instance, the user can specify that the atom $var(X)$ should be fully evaluated only if its argument is ground or if its argument is guaranteed to be free (at evaluation time) and that it should be residualised otherwise.

6. Experiments and results

6.1. An example: Comparison with LST

Before showing the results of our method, let us first illustrate in what sense it sometimes *improves* upon the method of Lloyd, Sonenberg and Topor (LST) in [61].

Example 6.1. Let the rules in Fig. 7 form the intensional part of $Db^{=}$ and let $Db^{+} = \{man(a) \leftarrow\}$, $Db^{-} = \emptyset$. Then, independently of the facts in $Db^{=}$, we have:

$$pos(U) = \{man(a),\ father(a,\_),\ married\_to(a,\_),\ married\_man(a),\ married\_woman(\_),\ unmarried(a),\ false\},$$

$$neg(U) = \{unmarried(a), false\}.$$

The LST method of [61] will then generate the following simplified integrity constraints:

$$false \leftarrow man(a), woman(a)$$

$$false \leftarrow parent(a, Y), unmarried(a)$$

Given the available information, this simplification of the integrity constraints is not optimal. Suppose that some fact matching $parent(a, Y)$ exists in the database. Evaluating the second simplified integrity constraints above, then leads to the incomplete SLDNF-tree depicted in Fig. 8 and subsequently to the evaluation of the goal:

$$\leftarrow woman(a), \neg married\_woman(a)$$

This goal is not *potentially added* and the derivation leading to the goal is not *incremental*. Hence, by Theorem 2.9, this derivation can be pruned and will never lead to a successful refutation, given the fact that the database was consistent before the update. The *incremental_solve* meta-interpreter of Appendix A improves upon this and does not evaluate $\leftarrow woman(a), \neg married\_woman(a)$.

$$\begin{array}{l}
mother(X,Y) \leftarrow parent(X,Y), woman(X) \\
father(X,Y) \leftarrow parent(X,Y), man(X) \\
grandparent(X,Z) \leftarrow parent(X,Y), parent(Y,Z), \\
married_to(X,Y) \leftarrow \\
\quad parent(X,Z), parent(Y,Z), \\
\quad man(X), woman(Y) \\
married_man(X) \leftarrow married_to(X,Y) \\
married_woman(X) \leftarrow married_to(Y,X) \\
unmarried(X) \leftarrow man(X), \neg married_man(X) \\
unmarried(X) \leftarrow woman(X), \neg married_woman(X) \\
\\
false \leftarrow man(X), woman(X) \\
false \leftarrow parent(X,Y), parent(Y,X) \\
false \leftarrow parent(X,Y), unmarried(X)
\end{array}$$

Fig. 7. Intensional part of $Db^{=}$.

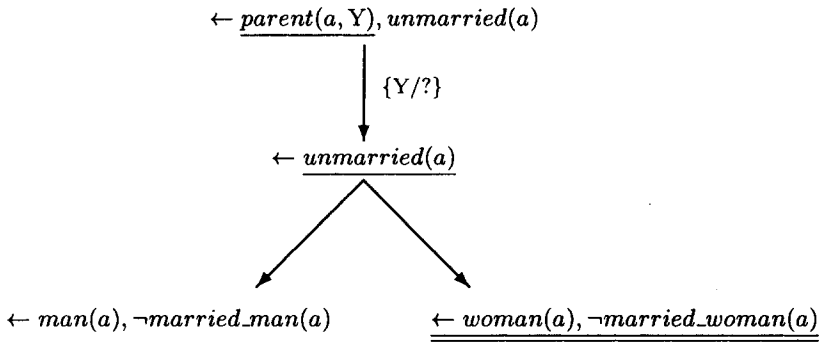

Fig. 8. SLDNF-tree for Example 6.1.

Actually, by partial evaluation of the *incremental_solve* meta-interpreter, this useless branch is already pruned at specialisation time. For instance, when generating a specialised update procedure for the pattern $Db^+ = \{man(\mathscr{A}) \leftarrow\}$, $Db^- = \emptyset$, we obtain the update procedure presented in Fig. 9. [17] This update procedure is very satisfactory and is in a certain sense optimal. The only way to improve it, would be to add the information that the predicates in the intensional and the extensional database are disjoint. For most applications this is the case, but it is not required by the current method. This explains the seemingly redundant test in Fig. 9, checking whether there is a fact *married_to* in the database. Benchmarks concerning this example will be presented in Section 6.2.

Finally, we show that, although the LST method of [61] can also handle update patterns, the simplified integrity constraints that one obtains in that way are neither very interesting nor very efficient. Indeed, one might think that it is possible to obtain specialised update procedures immediately from the LST method of [61], by running it on a generic update instead of a fully specified concrete update. Let us re-examine Example 6.1. For the generic update (pattern) $Db^+ = \{man(X) \leftarrow\}$, [18] $Db^- = \emptyset$, we obtain, independently of the facts in $Db^=$, the sets:

$$pos(U) = \{man(\_),\ father(\_,\_),\ married\_to(\_,\_),\ married\_man(\_),\ married\_woman(\_),\ unmarried(\_), false\},$$

$$neg(U) = \{unmarried(\_), false\}.$$

The LST method of [61] will thus generate the following simplified integrity constraints:

$$false \leftarrow man(X), woman(X)$$

[17] The figure actually contains a slightly sugared and simplified version of the resulting update procedure. There is no problem whatsoever, apart from finding the time for coding, to directly produce the sugared and simplified version. Also, all the benchmarks were executed on un-sugared and un-simplified versions.

[18] The LST method of [61] (as well as any other specialised integrity checking method we know of) cannot handle patterns of the form $man(\mathscr{A})$, where $\mathscr{A}$ represents an as of yet unknown constant. We thus have to use the variable X to represent the update pattern (replacing $\mathscr{A}$ by a constant would, in all but the simplest cases, lead to incorrect results).

```
incremental_solve__1(X1) :-
    fact(woman,[struct(X1,[])]).
incremental_solve__1(X1) :-
    fact(parent,[struct(X1,[]),X2]),
    (fact(married_to,[struct(X1,[]),X3])
     -> fail
     ;
      ((fact(parent,[struct(X1,[]),X4]),
        fact(parent,[X3,X4]),
        fact(woman,[X3]) )
        -> fail
        ; true
       )
     ).
```

Fig. 9. Specialised update procedure for adding $man(\mathscr{A})$.

$$false \leftarrow parent(X, Y), unmarried(X)$$

So, by running LST on a generic update we were just able to deduce that the second integrity constraint cannot be violated by the update – the other integrity constraints remain unchanged. This means that the specialised update procedures we obtain by this technique will in general only be slightly faster than fully re-checking the integrity constraints after each update. For instance, the above procedure will run (cf. Table 1 in the next subsection) considerably slower than the specialised update procedure in Fig. 9 obtained by LEUPEL. So, although the pre-compilation process for such an approach will be very fast, the obtained update procedures usually have little practical value. In order for this approach to be more efficient, the LST method should be adapted so that it can distinguish between variables and unknown input stemming from the update pattern. But this is exactly what our approach based on partial evaluation of meta-interpreters achieves!

6.2. Comparison with other partial evaluators

In this subsection, we perform some experiments with the database of Example 6.1. The goal of these experiments is to compare the annotation based partial evaluation technique presented in Section 5 with some existing *automatic* partial evaluators for full Prolog and give a first impression of the potential of our approach. In Subsection 6.3, we will do a more extensive study on a more complicated database,

Table 1
Results for $Db^- = \{man(\mathscr{A}) \leftarrow\}$, $Db^- = \emptyset$

solve	*ic-solve*	*ic-leupel*	*ic-leupel*$^-$	*ic-mixtus*	*ic-paddy*
108 facts:					
42.93 s	6.81 s	0.075 s	0.18 s	0.34 s	0.27 s
572.4	90.8	1	2.40	4.53	3.60
216 facts:					
267.9 s	18.5 s	0.155 s	0.425 s	0.77 s	0.62 s
1728.3	119.3	1	2.74	4.96	4.00

with more elaborate transactions, and show the benefits compared to the LST method [61].

The times for the benchmark are expressed in seconds and were obtained by calling the *time*/2 predicate of Prolog by BIM, which incorporates the time needed for garbage collection, see [80]. We used sets of 400 updates and a fact database consisting of 108 facts and 216 facts respectively. The rule part of the database is presented in Fig. 7. Also note that, in trying to be as realistic as possible, the fact part of the database has been simulated by Prolog facts (which are, however, still extracted a tuple at a time by the Prolog engine). The tests were executed on a Sun Sparc Classic running under Solaris 2.3.

The following different integrity checking methods were benchmarked:

1. *solve:* This is the naive meta-interpreter of Fig. 5. It does not use the fact that the database was consistent before the update and simply tries to find a refutation for $\leftarrow$ *false*.
2. *ic-solve:* This is the *incremental_solve* meta-interpreter performing specialised integrity checking, as described in Section 3. The skeleton of the meta-interpreter can be found in Fig. 6, the full code is in Appendix A.
3. *ic-leupel:* These are the specialised update procedures obtained by specialising *ic-solve* with the partial evaluation system LEUPEL described in Section 5. A prototype, based on LEUPEL, performing these specialisations fully automatically, is publicly available in [49]. This prototype can also be used to get the timings for *solve* and *ic-solve* above as well as *ic-leupel*$^-$ below.
4. *ic-leupel*$^-$*:* These are also specialised update procedures obtained by LEUPEL, but this time with the safe left-propagation of bindings (see Section 5.3.1) disabled.
5. *ic-mixtus:* These specialised update procedures were obtained by specialising *ic-solve* using the automatic partial evaluator MIXTUS described in [82,83]. Version 0.3.3 of MIXTUS, with the default parameter settings, was used in the experiments.
6. *ic-paddy:* These specialised update procedures were obtained by specialising *ic-solve* using the automatic partial evaluator PADDY presented in [76–78]. The resulting specialised procedures had to be slightly converted for Prolog by BIM: `get_cut/1` had to be transformed into `mark/1` and `cut_to/1` into `cut/1`. We also had to increase the "term_depth" parameter of PADDY from its default value. With the default value, PADDY actually slowed down the *ic-solve* meta-interpreter by about 30%.

The first experiment we present consists in generating an update procedure for the update pattern:

Table 2
Results for $Db^+ = \{parent(\mathscr{A}, \mathscr{B}) \leftarrow\}$, $Db^- = \emptyset$

solve	*ic-solve*	*ic-leupel*	*ic-leupel*$^-$	*ic-mixtus*	*ic-paddy*
108 facts:					
43.95 s	7.75 s	0.24 s	0.355 s	0.53 s	0.45 s
183.1	32.3	1	1.48	2.21	1.88
216 facts:					
273.1 s	21.9 s	0.915 s	1.16 s	1.67 s	1.435 s
298	23.9	1	1.26	1.82	1.57

$$Db^+ = \{man(\mathscr{A}) \leftarrow\}, \qquad Db^- = \emptyset,$$

where $\mathscr{A}$ is unknown at partial evaluation time. The result of the partial evaluation obtained by LEUPEL can be seen in Fig. 9 and the timings are summarised in Table 1. The first row of figures contains the absolute and relative times required to check the integrity for a database with 108 facts. The second row contains the corresponding figures for a database with 216 facts.

The times in Table 1 (as well as in Table 2) include the time to specialise the integrity constrains as well as the time to run them. Note that *solve* performs no specialisation, while all the other methods interleave execution and specialisation, as explained in Section 3. The times required to generate the specialised update procedures (*ic-leupel*, *ic-leupel*$^-$, *ic-mixtus* and *ic-paddy*) for the above update pattern are not included. In fact these update procedures only have to be regenerated when the rules or the integrity constraints change. The time needed to obtain the *ic-leupel* specialised update procedure was 78.19 s. The current implementation of LEUPEL has a very slow post-processor, displays tracing information and uses the ground representation. Therefore, it is certainly possible to reduce the time needed for partial evaluation by at least one order of magnitude. Still, even using the current implementation, the time invested into partial evaluation should pay off rather quickly for larger databases.

In another experiment we generated a specialised update procedure for the following update pattern:

$$Db^+ = \{parent(\mathscr{A}, \mathscr{B}) \leftarrow\}, \quad Db^- = \emptyset,$$

where $\mathscr{A}$ and $\mathscr{B}$ are unknown at partial evaluation time. This update pattern offers less opportunities for specialisation than the previous one. The speedup figures are still satisfactory but less spectacular. The results are summarised in Table 2.

In summary, the speedups obtained with the LEUPEL system are very encouraging. The specialised update procedures execute up to 2 orders of magnitude faster than the intelligent incremental integrity checker *ic-solve* and up to 3 orders of magnitude faster than the non-incremental *solve*. The latter speedup can of course be made to grow to almost any figure by using larger databases. Note that, according to our experience, specialising the *solve* meta-interpreter of Fig. 5 usually yields speedups reaching at most 1 order of magnitude.

Also, the specialised update procedures obtained by using the LEUPEL system performed between 1.6 and 5 times faster than the ones obtained by the fully automatic systems MIXTUS and PADDY. This shows that using annotations, combined with restricting the attention to ITE-Prolog instead of full Prolog, pays off in better specialisation. Finally, note that the safe left-propagation of bindings described in Section 5.3.1 has a definite, beneficial effect on the efficiency of the specialised update procedures.

6.3. A more comprehensive study

In this subsection, we perform a more elaborate study of the specialised update procedures generated by LEUPEL and compare their efficiency with the one of the LST techniques [61], which often performs very well in practice, even for relational or hierarchical databases [22]. To that end we will use a more sophisticated database

Table 3
Results for $Db^+ = \{father(\mathscr{X},\mathscr{Y}) \leftarrow\}$, $Db^- = \emptyset$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts:					
1	{2,2}	33.20 s	18.72 s	0.07 s	6.62 s
		474	267	1	95
2	{8}	32.60 s	18.35 s	0.04 s	6.51 s
		815	262	1	163
3	{ }	33.10 s	18.54 s	0.07 s	6.51 s
		473	265	1	93
238 facts:					
1	{2,2}	69.60 s	32.56 s	0.13 s	6.75 s
		535	250	1	52
2	{8}	68.30 s	32.40 s	0.10 s	6.51 s
		683	324	1	65
3	{ }	67.90 s	31.90 s	0.13 s	6.51 s
		522	245	1	50

and more complicated transactions. The rules and integrity constraints $Db^=$ of the database are taken from [84] and can be found in Appendix C.

For the benchmarks of this subsection, *solve*, *ic-solve* and *ic-leupel* are the same as in the Subsection 6.2. In addition we also have the integrity checking method *ic-lst*, which is an implementation of the LST method [61] and whose code can be found in Appendix B. [19]

For the more elaborate benchmarks, we used five different update patterns. The results are summarised in the Tables 3–7. The particular update pattern, for which the specialised update procedures were generated, can be found in the table description. Different concrete updates, all instances of the given update pattern, were used to measure the efficiency of the methods. The second column of each table contains the integrity constraints violated by each concrete update. For each particular concrete update, the first row contains the absolute times for 100 updates and the second row contains the relative time w.r.t. LEUPEL. Each table is divided into sub-tables for databases of different sizes (and in case of *ic-leupel* the *same* specialised update procedure was used for the different databases). As in the previous experiments, the times to simplify and run the integrity constraints, given the concrete update, were included. The time to generate the specialised updated procedures (*ic-leupel*) is not included. As justified in Section 6.1, *ic-lst* is run on the concrete update and not on the update pattern.

As can be seen from the benchmark tables, the update procedures generated by LEUPEL perform extremely well. In Table 6, LEUPEL detected that there is no way this update can violate the integrity constraints – hence the "infinite" speedup. For 238 facts, the speedups in the other tables range from 99 to 918 over *solve*, from

[19] We also tried a "dirty" implementation using assert and retracts to store the potential updates. But, somewhat surprisingly, this solution ran slower than the one shown in Appendix B which stores the potential updates in a list.

Table 4
Results for $Db^+ = \{civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$, $Db^- = \emptyset$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
128 facts:					
1	{5,6}	28.80 s	35.40 s	0.17 s	13.12 s
		169	208	1	77
2	{a1, a1, 5, 10}	29.70 s	37.21 s	0.34 s	12.58 s
		87	109	1	37
3	{ }	28.70 s	36.50 s	0.26 s	11.69 s
		110	140	1	45
238 facts:					
1	{5, 6}	67.50 s	77.61 s	0.30 s	12.88 s
		225	259	1	43
2	{a1, a1, 5, 10}	68.00 s	80.20 s	0.69 s	12.66 s
		99	116	1	18
3	{ }	67.30 s	80.49 s	0.55 s	11.76 s
		122	145	1	21

Table 5
Results for $Db^+ = \{father(\mathcal{F}, \mathcal{X}), civil\_status(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{T}) \leftarrow\}$, $Db^- = \emptyset$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts:					
1	{ }	36.30 s	47.80 s	0.21 s	19.68 s
		173	228	1	94
238 facts:					
2	{ }	78.10 s	95.30 s	0.41 s	20.32 s
		190	232	1	50

Table 6
Results for $Db^+ = \emptyset$, $Db^- = \{father(\mathcal{X}, \mathcal{Y}) \leftarrow\}$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts:					
1	{ }	59.60 s	3.50 s	0.00 s	6.50 s
		"∞"	"∞"	1	"∞"
2	{ }	59.40 s	3.50 s	0.00 s	6.54 s
		"∞"	"∞"	1	"∞"
238 facts:					
1	{ }	59.70 s	3.50 s	0.00 s	6.55 s
		"∞"	"∞"	1	"∞"
2	{ }	59.10 s	3.60 s	0.00 s	6.53 s
		"∞"	"∞"	1	"∞"

Table 7
Results for $Db^+ = \emptyset$, $Db^- = \{civil\_status(\mathscr{X}, \mathscr{Y}, \mathscr{Z}, \mathscr{T}) \leftarrow\}$

Update	ICs viol	*solve*	*ic-solve*	*ic-leupel*	*ic-lst*
138 facts:					
1	{8,8,9a}	59.90 s	13.45 s	0.18 s	16.60 s
		333	75	1	92
2	{8,8,9a}	59.75 s	11.25 s	0.15 s	12.08 s
		398	75	1	81
3	{ }	60.70 s	11.00 s	0.08 s	11.96 s
		759	69	1	150
4	{ }	59.90 s	13.50 s	0.11 s	16.47 s
		545	123	1	150
238 facts:					
1	{8,8,9a}	74.10 s	17.50 s	0.19 s	17.38 s
		390	92	1	91
2	{8,8,9a}	73.10 s	14.30 s	0.16 s	12.64 s
		457	89	1	67
3	{ }	73.50 s	14.00 s	0.08 s	12.56 s
		918	175	1	157
4	{ }	72.60 s	17.20 s	0.11 s	17.25 s
		660	156	1	157
338 facts:					
1	{8,8,9a}	114.20 s	22.30 s	0.28 s	17.09 s
		407	80	1	61
438 facts:					
1	{8,8,9a}	161.60 s	27.30 s	0.36 s	17.06 s
		448	76	1	47
838 facts:					
1	{8,8,9a}	391.50 s	48.10 s	0.68 s	17.26 s
		576	71	1	25

89 to 324 over *ic-solve* and from 18 to 157 over *ic-lst*. These speedups are very encouraging and lead us to conjecture that the approach presented in this paper can be very useful in practice and lead to big efficiency improvements.

Of course, the larger the database becomes, the more time will be needed on the actual evaluation of the simplified constraints and not on the simplification. That is why the relative difference between *ic-lst* and *ic-leupel* diminishes with a growing database. However, even for 838 facts in Table 7, *ic-leupel* still runs 25 times faster than *ic-lst*. So for all examples tested so far, using an evaluation mechanism which is slower than in "real" database systems and therefore exaggerates the effect of the size of the database on the benchmark figures (but is still tuple-oriented – future work will have to examine how the current technique and experiments carry over to a set-oriented environment), the difference remains significant.

We also measured heap consumption of *ic-leupel*, which used from 43 to 318 times less heap space than *ic-solve*. Finally, in a small experiment we also tried to specialise *ic-lst* for the update patterns using MIXTUS, but without much success. Speedups were of the order of 10%.

7. Conclusion and future directions

7.1. Conclusion and discussion

We presented the idea of obtaining specialised update procedures for deductive databases in a principled way: namely by writing a meta-interpreter for specialised integrity checking and then partially evaluating this meta-interpreter for certain update patterns. The goal was to obtain specialised update procedures which perform the integrity checking much more efficiently than the generic integrity checking methods.

Along the way we have gained insights into issues concerning the ground versus the non-ground representation. We notably argued for the use of a "mixed" representation, in which the object program is represented using the ground representation, but where the goals are lifted to the non-ground representation for resolution. This approach has the advantage of using the flexibility of the ground representation for representing knowledge about the object program (in our case the deductive database along with the updates), while using the efficiency of the non-ground representation for resolution. The mixed representation is also much better suited for partial evaluation than the full ground representation.

We have also argued why some existing integrity checking methods, like, e.g., the Lloyd–Sonenberg–Topor method [61], are not amenable to specialisation using current techniques. The paper therefore presents a new integrity checking method which, in the context of *hierarchical* databases, can be implemented in the mixed representation and which is well suited for partial evaluation. However, for efficiency reasons, the implementation still had to make use of a non-declarative *verify* construct. In essence, the *verify* construct is used to test whether a given goal, encountered while checking the integrity of a database upon an update, might be influenced by that update. If this is not the case the integrity checker will stop the derivation of the goal.

This *verify* primitive can be implemented via the if-then-else construct. We have then presented an extension of pure Prolog, called ITE-Prolog, which incorporates the if-then-else and we have presented how this language can be specialised. We have drawn upon the techniques in [48] and presented the partial evaluator LEUPEL and a prototype [49] based upon it, which can generate specialised update procedure fully automatically.

This prototype has been used to conduct extensive experiments, the results of which were very encouraging. Speedups reached and exceeded 2 orders of magnitude when specialising the integrity checker for a given set of integrity constraints and a given set of rules. These high speedups are also due to the fact that the partial evaluator performs part of the integrity checking. We also compared the specialised update procedures with an unspecialised implementation of the well known approach by Lloyd et al. in [61], and the results show that big performance improvements, also reaching and exceeding 2 orders of magnitude, can be obtained.

Within this paper, we have restricted our attention to hierarchical databases. Our long term objective is of course to move to recursive databases, but as pointed out in Section 4 this is far from obvious (progress has already been made, e.g., in [54–56,51,58]). Still, one could argue that our experimental results should therefore not have been compared with alternative approaches for integrity checking in deduc-

tive databases, but compared to those for relational databases instead. However, even for relational databases – with complex views – the deductive database approaches, such as the LST method [61], seem to be the most competitive ones available. The simpler alternative of mapping down intensional database relations to extensional ones (through full unfolding of the views) and performing relational integrity checking on the extensional database predicates, tends to create extensive redundant multiplication of checks.

Example 7.1. Let us try to fully unfold the integrity constraints of the simple database in Fig. 7. Note that in general, unfolding inside negation is tricky. Here however, we are lucky as all negated rules are just defined by 1 clause, and full unfolding will give:

$$false \leftarrow man(X), woman(X)$$

$$false \leftarrow parent(X,Y), parent(Y,X)$$

$$false \leftarrow parent(X,Y), man(X), \neg parent(X,\_)$$

$$false \leftarrow parent(X,Y), man(X), \neg parent(\_Z,\_)$$

$$false \leftarrow parent(X,Y), man(X), \neg man(X)$$

$$false \leftarrow parent(X,Y), man(X), \neg woman(\_Z)$$

$$false \leftarrow parent(X,Y), woman(X), \neg parent(\_Z,\_)$$

$$false \leftarrow parent(X,Y), woman(X), \neg parent(X,\_)$$

$$false \leftarrow parent(X,Y), woman(X), \neg man(\_Z)$$

$$false \leftarrow parent(X,Y), woman(X), \neg woman(X)$$

So even for this rather simple example, the fully unfolded integrity constraints become quite numerous.

For the update $Db^+ = \{man(a)\}$, $Db^- = \emptyset$ we then get, by unifying the update literals with the body atoms, the following specialised integrity constraints:

$$false \leftarrow woman(a)$$

$$false \leftarrow parent(a,Y), \neg parent(a,\_)$$

$$false \leftarrow parent(a,Y), \neg parent(\_Z,\_)$$

$$false \leftarrow parent(a,Y), \neg man(a)$$

$$false \leftarrow parent(a,Y), \neg woman(\_Z)$$

Note that the $parent(a,Y)$ has been duplicated 4 times! The resulting simplified integrity checks are thus not nearly as efficient as our specialised update procedure in Fig. 9 (especially if a has a lot of children), which executes this $parent(a,Y)$ only once. For more complicated examples, entire conjunctions will get duplicated, making the situation worse. This is the price one has to pay for getting rid of the rules:

rules capture common computations and avoid that they get repeated (memorisation will only solve this problem for atomic queries).

As such, even in the context of hierarchical databases, our comparisons are with respect to the best alternative approaches [22].

Finally, as compile-time optimisation is a trade-off between run-time and compile-time execution, one could argue that the specialisation times should have been considered more carefully in our evaluation of the approach.

However, note that for integrity checking in databases

1. The frequency with which updates (and integrity checks) are performed is significantly higher than that of remodelling (and specialising) [20] the static part of the database. In general, it is difficult to evaluate the real efficiency gain, as it is very dependent on the ratio of these two. On the other hand, if the ratio is sufficiently high, specialisation times become less relevant.
2. Updates are often performed interactively, on-line, making acceptably fast execution important, while respecialisation would typically be performed in background, requiring less efficiency.

For these reasons, we devoted less attention to the efficiency of the specialisation phase, for which we make use of a prototype specialiser. Still, one might consider to use, e.g., the work by Benkerimi and Shepherdson [5] to *incrementally* adapt the specialised update procedures whenever the rules or integrity constraints change. Another approach might be based on using a *self-applicable* partial evaluation system in order to obtain efficient update procedure compilers by self-application.

To summarise, we have shown that, given proper care in writing a meta-interpreter, partial evaluation is capable of automatically generating highly specialised update procedures for hierarchical databases with negation.

7.2. Future directions

In future work, one might apply the techniques of this paper to other meta-interpreters, which have a more flexible way of specifying static and dynamic parts of the database and are less entrenched in the concept that facts change more often than rules and integrity constraints.

On the level of practical applications, one might try to apply the methods of this paper to abductive and inductive logic programs. For instance, we conjecture that solvers for abduction, like the SLDNFA procedure [24], can greatly benefit in terms of efficiency, by generating specialised integrity checking procedures for each abducible predicate.

Finally, it might also be investigated whether partial evaluation *alone* is able to derive specialised integrity checks. In other words, is it possible to obtain specialised integrity checks by specialising a simple solve meta-interpreter, like the one of Fig. 5. In that case, the assumption that the integrity constraints were satisfied before the update has to be handed to the specialiser, for instance in the form of a constraint. The framework of constrained partial deduction, developed in [52], could be used to

[20] For the examples presented in this paper, the update procedures have to be re-generated when the rules or the integrity constraints change (but not when the facts change).

that effect. In such a setting, self-applicable constrained partial deduction could be used to obtain specialised update procedures by performing the second Futamura projection [28,26] and update procedure compilers by performing the third Futamura projection.

Acknowledgements

Michael Leuschel was supported by Esprit BR-project Compulog II and the Belgian GOA "Non-Standard Applications of Abstract Interpretation". Danny De Schreye is senior research associate of the Belgian National Fund for Scientific Research. We would like to thank Bern Martens for proof-reading several versions of this paper and for his helpful insights and comments on the topic of this paper. We would also like to thank him for his huge pile of references on integrity checking and for introducing the first author to the subject. We thank Bart Demoen for sharing his expertise on writing efficient Prolog programs. Our thanks also go to John Gallagher for pointing out several errors in an earlier version of the paper and for the fruitful discussions on partial evaluation and integrity checking. We are also grateful for discussions and extremely helpful comments by Hendrik Decker. Other interesting discussions about the topic of this paper were held with Stefan Decker, the members of the Compulog II project as well as with the participants of the 1996 Dagstuhl seminar on "Logic and the Meaning of Change". Finally we would like to thank all the anonymous referees for their very useful remarks and for helping to clarify the paper.

Appendix A. The ic-solve meta-interpreter

This appendix contains the full Prolog code of a meta-interpreter performing incremental integrity checking based upon Theorem 2.9.

In order to make the experiments more realistic, the facts of the database are stored, via the `fact/2` relation, in the Prolog clausal database. Updating the facts – which is of no relevance to the integrity checking phases – occurs via the `assert`, `retract` or `reconsult` primitives.

```
/* ------------------------------------------------------------ */
/* normal_solve(GrXtrFacts.,GrDelFacts,GrRules,NgGoal) */
/* ------------------------------------------------------------ */

/* This normal_solve makes no assumptions about the Facts and the Rules.
For instance the predictates defined in Rules can also be present in Facts
and vice versa */

normal_solve(GrXtraFacts,GrDelFacts,GrRules,[]).
normal_solve(GrXtraFacts, GrDelFacts,GrRules,[neg(NgG)| NgT]):-
   (normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos[NgG)])
   → fail
   ;  (normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT))
   ).
```

```
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH) | NgT]):-
   db_fact_lookup(NgH),
   non(non_ground_member(NgH,GrDelFacts)),
   normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH) | NgT]):-
   non_ground_member(NgH,GrXtraFacts),
   normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).
normal_solve(GrXtraFacts,GrDelFacts,GrRules,[pos(NgH) | NgT]):-
   non_ground_member(term(clause,[pos(NgH)|NgBody]),GrRules),
   normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgBody),
   normal_solve(GrXtraFacts,GrDelFacts,GrRules,NgT).

/* ------------------------- */
/* INCREMENTAL IC CHECKER */
/* ------------------------- */

incremental_solve(GoalList,DB):-
   verify_one_potentially_added(GoalList,DB),
   inc_resolve(GoalList,DB).

inc_resolve([pos(NgH) | NgT],DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOldRules,AddedRules,DeletedRules),
   dt_fact_lookup(NgH),
   not(non_ground_member(NgH,DeletedFacts)),
   incremental_solve(NgT,DB).
inc_resolve([pos(NgH)|NgT],DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOlRules,AddedRules,DeletedRules),
   non_ground_member(NgH,AddedFacts),
   /* print (found_added_fact(NgH)),nl. */
   append(AddedRules,ValidOldRules,NewRules),
   normal_solve(AddedFacts,DeletedFacts,NewRules,NgT).
inc_resolve([pos(NgH)|NgT],DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOldRules,AddedRules,DeletedRules)
   non_ground_member(term(clause,[pos(NgH)|NgBody]),ValidOldRules),
   append(NgBody,NgT,NewGoal),
   incremental_solve(NewGoal,DB).
inc_resolve([pos(NgH)|NgT),DB:-
   DB = db(AddedFacts,DeletedFacts,
   ValidOlRules,AddedRules,DeletedRules),
   non_ground_member(term(clause,[pos(NgH)|NgBody]),AddedRules),
   append(AddedRules,ValidOlRules,NewRules),
   normal_solve(AddedFacts,DeletedFacts,NewRules,NgBody),
   normal_solve(AddedFacts,DeletedFacts,NewRules,NgT).
inc_resolve([neg(NgH)|NgT],DB):-
   DB = db(AddedFacts,DeletedFacts,
```

```
      ValidOldRules,AddedRules,DeletedRules),
   append(AddedRules,ValidOlRuleds,NewRules),
   (normal_solve(AddedFacts,DeletedFacts,NewRules,[pos(NgH)])
   → (fail)
   ;  (verify_potentially_added(neg(NgH),DB)
    → (normal_solve(AddedFacts,DeletedFacts,NewRules,NgT))
   ;  (incremental_solve(NgT,DB))
    )
   ).

verify_one_potentially_added(GoalList,DB):-
   ( (one_potentially_added(GoalList,DB) → fail; true)
   → fail
   ;  true
   ).

one_potentially_added(GoalList,DB):-
   member(Literal,GoalList),
   potentially_added(Literal,DB).

/* ------------------------------------------------------------ */
/* Determining the literals that are potentially added */
/* ------------------------------------------------------------ */

/* verify if a literal is potentially added -
   without making any bindings and succeeding only once */
verify_potentially_added(Literal,DB):-
   ( (potentially_added(Literal,DB) → fail; true)
   → fail
   ;  true
   ).

potentially_added(neg(Atom),DV):-
   potentially_deleted(pos(Atom),DB).
potentially_added(pos(Atom),DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOldRules,AddedRules,DeletedRules),
   non_ground_member(Atom,AddedFacts).
potentially_added(pos(Atom),DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOldRules,AddedRules,DeletedRules),
   non_ground_member(term(clause,[pos(Atom)|NgBody]),AddedRules).
potentially_added(pos(Atom),DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOlRules,AddedRules,DeletedRules),
   non_ground_member(term(clause,[pos(Atom)|NgBody]),ValidOldRules),
   member(BodyLiteral,NgBody),
   potentially_added(BodyLiteral,DB).
```

```
potentially_deleted(neg(Atom),DB):-
   potentially_added(pos(Atom),DB).
potentially_deleted(pos(Atom),DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOldRules,AddedRules,DeletedRules).
   non_ground_member(Atom,DeletedFacts).
potentially_deleted(pos(Atom),DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOldRules,AddedRules,DeletedRules),
   non_ground_member(term(clause,[pos(Atom)|NgBody]),DeletedRules).
potentially_deleted(pos(Atom),DB):-
   DB = db(AddedFacts,DeletedFacts,
   ValidOldRules,AddedRules,DeletedRules),
   non_ground_member(term(clause,[pos(Atom)|NgBody]),ValidOldRules),
   member(BodyLiteral,NgBody),
   potentially_deleted(BodyLiteral,DB).

/* ------------------------------------------------ */
/* non_ground_member(NgExpr,GrListOfExpr) */
/* ------------------------------------------------ */

non_ground_member(NgX,[GrH|GrT]):-
   make_non_ground(GrH,NgX).
non_ground_member(NgX,[GrH|GrT]):-
   non_ground_member(NgX,GrT).

/* ------------------------------------------------------------ */
/*  make_non_ground(GroundRepOfExpr,NonGroundRepOfExpr)  */
/* ------------------------------------------------------------ */
/* ex. ?-make\_non\_groung(pos(term(f,[var(1), var(2), var(1)])),X). */

make_non_ground(G,NG):-
   mng(G,NG,[],Sub).

mng(var(N),X,[],[sub(N,X)]).
mng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|Tl]):-
   ((N=M)
   -> (Tl=T, X=Y)
   ;  (mng(var(N),X,T,Tl))
   ).
mng(term(F,Args),term(F,IArgs),InSub,OutSub):-
   l_mng(Args,IArgs,InSub,OutSub).
mng(neg(G),neg(NG),InSub,OutSub):-
   mng(G,NG,InSub,OutSub).
mng(pos(G),pos(NG),InSub,OutSub):-
   mng(G,NG,InSub,OutSub).

l_mng([],[],Sub,Sub).
```

```
l_mng([H|T],([IH|IT],InSub,OutSub):-
  mng(H,IH,InSub,InSub),
  l_mng(T,IT,InSub,OutSub).

/* ------------------------------------------------------ */
/* SIMULATING THE DEDUCTIVE DATABASE FACT LOOKUP */
/* ------------------------------------------------------ */

db_fact_lookup(term(Pred,Args)):-
  fact(Pred,Args).
fact(female,[term(mary,[])]).
fact(male,[term(peter,[])]).
fact(male,[term(paul,[])]).
```

Appendix B. The ic-lst meta-interpreter

This appendix contains the code of an implementation of the method by Lloyd et al. [61] for specialised integrity checking in deductive databases.

```
/*=========================================================*/
/* Bottom-Up Propagation of update according to Lloyd et al's Method */
/*=========================================================*/

:- dynamic lts_rules/1.

construct_lts_rules :-
  retract(lts_rules(R)),fail.
construct_lts_rules :-
  findall(clause(Head,Body),rule(Head,Body),Rules),
  assert(lts_rules(Rules)).

lts_check(Nr.Update) :-
  lts_rules(Rules),
  check_ic(Nr,Update,Rules).

check_ic(Nr,Update,Rules) :-
  bup(Rules,Update,AllPos),!,
  member(false(Nr),AllPos),
  member(clause(false(Nr),Body),Rules),
  member(Atom,Body),
  member(Atom,AllPos),
  normal_solve(Body,Update).

/* This is the main Predicate */
/* Rules is the intensional part of the database */
/* Update are the added facts to the extensional database */
/* Pos is the set of (most general) atoms potentially */
/*  affected by the update */
```

```
bup(Rules,Update,Pos) :-
  bup(Rules, Update, Update,Pos).

bup(Rules, Update, InPos, OutPos) :-
  bup_step(Rules,Update,[],NewPos,InPos,IntPos),

  ((NewPos=[])
  → (OutPos=IntPos)
  ;  (bup(Rules,NewPos,IntPos,Outpos))
  ).

bup_step([],_Pos,NewPos,NewPos,AllPos,AllPos).
bup_step([Clause|Rest],Pos,InNewPos,ResNewPos,InAllPos,ResAllPos) :-
  Clausel = clause(Head,Body),
  bup_treat_clause(Head,Body,Pos,InNewPos1,InAllPos,InAllPos1),
  bup_step(Rest,Pos,InNewPos1,ResNewPos,InAllPos1,ResAllPos).

bup_treat_clause(Head,[],Pos,NewPos,NewPos,AllPos,AllPos).
bup_treat_clause(Head,[BodyAtom|Rest],Pos,InNewPos,OutNewPos,
      InAllPos,OutAllPos) :-
  bup-treat_body_atom(Head,BodyAtom,Pos,InNewPos,InNewPos1,
      InAllPos,InAllPos1).
  bup_treat_clause(Head,Rest,Pos,InNewPos1,OutNewPos,
      InAllPos1,OutAllPos).

bup_treat_body_atom(Head,BodyAtom,[],NewPos,NewPos,AllPos,AllPos).
bup_treat_body_atom(Head,BodyAtom,[Pos1|Rest],InNewPos,OutNewPos,
       InAllPos,OutAllPos):-
  copy(Pos1,Pos1C),
  copy(g(Head,BodyAtom),g(CHead,CBodyAtom)),
  (propagate_atom(CHead,CBodyAtom,Pos1C,NewHead)
  → (add_atom(NewHead,InAllPos,InAllPos1,Answer),
    ((Answer=dont_add)
    → (InNewPos2=InNewPos,InAllPos2=InAllPos1)
    ;  (add_atom(NewHead,InNewPos,InNewPos1,Answer2),
    ((Answer2=dont_add)
    → (InNewPos2=InNewPos1,InAllPos2=InAllPos1)
    ; (InNewPos2=[NewHead|InNewPos1].
    InAllPos2=[NewHead|InAllPos1]
    )
    )
   )
   )
  )
  ;  (InNewPos=InNewPos,InAllPos2=InAllPos)
  ),
  bup_treat_body_atom(Head,BodyAtom,Rest,InNewPos2,OutNewPos,
      InAllPos2,OutAllPos).
```

```
propagate_atom(Head,BodyAtom,Pos,NewAtom) :-
  BodyAtom = Pos, !,
  NewAtom = Head.
propagate_atom(Head,BodyAtom,neg(Pos),NewAtom) :-!,
  BodyAtom = Pos,
  NewAtom = neg(Head).
propagate_atom(Head,neg(BodyAtom) Pos,NewAtom) :-!,
  BodyAtom = Pos,
  NewAtom = neg(Head).

add_atom(NewAtom, [],[],add).
add_atom(NewAtom,[Posl|Rest],OutPos,Answer):-
  (covered(NewAtom,Posl)
  → (OutPos = [Posl|Rest],
    Answer=dont_add
  )
  ;  (covered(Posl,NewAtom)
  → (Outpos = OutRest,
  add_atom(NewAtom,Rest,OutRest,Answer)
    )
  ;  (OutPos=[Postl|OutRest],
  add_atom(NewAtom,Rest,OutRest,Answer)
    )
  )
  ).
```

Appendix C. A more sophisticated database

The following is the intensional part of a database adapted from [84] (where it is used for the most complicated benchmark) and transformed into rule format (using Lloyd–Topor transformations [62] done by hand) required by [49]. The symbol ~ denotes negation while & denotes conjunction.

```
parent(B,C) ← father(B,C)
parent(B,C) ← mother(B,C)

mother(B,C) ← father(D,C) & husband(D,B)

age(Id,Age) ← civil_status(Id,Age,D,E)

sex(Id,C) ← civil_status(Id,Age,C,E)

dependent(B,C) ← parent(C,B) & occupation(C,service) & occupation(B,student)

occupation(Id,C) ← civil_status(Id,D,E,C)

eq(B,B) ←
```

```
aux_male_female(male) ←
aux_male_female(female) ←
aux_status(student) ←
aux_status(retired) ←
aux_status(business) ←
aux_status(service) ←
aux_limit(Id,Age) ← greater_than(Id,0) & less_than(Id,100000) &
                    greater_than(Age,0) & less_than(Age,125)

false(a1) ← civil_status(Id,Age,D,E) & civil_status(Id,F,G,H) & ~eq(Age,F)
false(a2) ← civil_status(Id,Age,D,E) & civil_status(Id,F,G,H) & ~eq(D,G)
false(a3) ← civil_status(Id,Age,D,E) & civil_status(Id,F,G,H) & ~eq(E,H)
false(2) ← father(B,C) & father(D,C) & ~eq(B,D)
false(3) ← husband(B,C) & husband(D,C) & ~eq(B,D)
false(4) ← husband(B,C) & husband(B,D) & ~eq(C,D)
false(5) ← civil_status(Id,Age,D,E) & aux_male_female(D) &
           aux_status(E) & ~aux_limit(Id,Age)
false(6) ← civil_status(Id,Age,D,student) & ~less_than(Age,25)
false(7) ← civil_status(Id,Age,D,retired) & ~greater_than(Age,60)
false(8) ← father(B,C) & ~sex(B,male)
false(9a) ← husband(B,C) & ~sex(B,male)
false(9b) ← husband(B,C) & ~sex(C,female)
false(10a) ← husband(B,C) & age(B,D) & ~greater_than(D,19)
false(10b) ← husband(B,C) & age(C,D) & ~greater_than(D,19)
false(11) ← civil_status(Id,Age,D,E) & less_than(Age,20) & ~eq(E,student)
false(12) ← dependent(X,Y) & ~tax(Y,X)
```

References

[1] L.O. Andersen, Partial evaluation of C and automatic compiler generation, in: U. Kastens, P. Pfahler (Eds.), Proceedings of the Fourth International Conference on Compiler Construction, Paderborn, Germany, Lecture Notes in Computer Science, vol. 641, Springer, Berlin, 1992, pp. 251–257.

[2] L.O. Andersen, Binding-time analysis and the taming of c pointers, in: Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, ACM Press, New York, 1993, pp. 47–58.

[3] K.R. Apt, Introduction to logic programming, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Chap. 10, North-Holland, Amsterdam, 1990, pp. 495–574.

[4] M. Baudinet, Proving termination of Prolog programs: A semantic approach, J. Logic Programming 14 (1/2) (1992) 1–29.

[5] K. Benkerimi, J.C. Shepherdson, Partial deduction of updateable definite logic programs, J. Logic Programming 18 (1) (1994) 1–27.

[6] R. Bol, Loop checking in partial deduction, J. Logic Programming 16 (1/2) (1993) 25–46.

[7] A. Bossi, N. Cocco, S. Etalle, Transformation of left terminating programs: The reordering problem, in: M. Proietti (Ed.), Logic Program Synthesis and Transformation, Proceedings of LOPSTR'95, Utrecht, The Netherlands, Lecture Notes in Computer Science, vol. 1048, Springer, Berlin, 1995, pp. 33–45.

[8] A.F. Bowers, Representing Gödel object programs in Gödel, Technical Report CSTR-92-31, University of Bristol, November 1992.

[9] A.F. Bowers, C.A. Gurr, Towards fast and declarative meta-programming, in: K.R. Apt, F. Turini (Eds.), Meta-logics and Logic Programming, MIT Press, Cambridge, MA, 1995, pp. 137–166.

[10] M. Bruynooghe, D. De Schreye, B. Martens, A general criterion for avoiding infinite unfolding during partial deduction, New Gener. Comput. 11 (1) (1992) 47–79.
[11] F. Bry, H. Decker, R. Manthey, A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases, in: J. Schmidt, S. Ceri, M. Missikoff (Eds.), Proceedings of the International Conference on Extending Database Technology, Venice, Italy, 1988, Lecture Notes in Computer Science, Springer, Berlin, 1988, pp. 488–505.
[12] F. Bry, R. Manthey, Tutorial on deductive databases, in: Logic Programming Summer School, 1990.
[13] F. Bry, R. Manthey, B. Martens, Integrity verification in knowledge bases, in: A. Voronkov (Ed.), Logic Programming, Proceedings of the First and Second Russian Conference on Logic Programming, Lecture Notes in Computer Science, vol. 592, Springer, Berlin, 1991, pp. 114–139.
[14] M. Bugliesi, F. Russo, Partial evaluation in Prolog: Some improvements about cut, in: E.L. Lusk, R.A. Overbeek (Eds.), Logic Programming: Proceedings of the North American Conference, MIT Press, Cambridge, MA, 1989, pp. 645–660.
[15] M. Celma, H. Decker, Integrity checking in deductive databases – the ultimate method? in: Proceedings of the Fifth Australasian Database Conference, January 1994.
[16] M. Celma, C. Garcí, L. Mota, H. Decker, Comparing and synthesizing integrity checking methods for deductive databases, in: Proceedings of the Tenth IEEE Conference on Data Engineering, 1994.
[17] C. Consel, A tour of Schism: A partial evaluation system for higher-order applicative languages, in: Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New York, 1993, pp. 145–154.
[18] C. Consel, O. Danvy, Tutorial notes on partial evaluation, in: Proceedings of ACM Symposium on Principles of Programming Languages (POPL'93), Charleston, South Carolina, ACM, New York, 1993.
[19] S. Das, M. Williams, A path finding method for constraint checking in deductive databases, Data Knowledge Eng. 4 (1989) 223–244.
[20] D. De Schreye, B. Martens, A sensible least Herbrand semantics for untyped vanilla meta-programming, in: A. Pettorossi (Ed.), Proceedings Meta'92, Lecture Notes in Computer Science, vol. 649, Springer, Berlin, 1992, pp. 192–204.
[21] H. Decker, Integrity enforcement on deductive databases, in: L. Kerschberg (Ed.), Proceedings of the First International Conference on Expert Database Systems, Charleston, South Carolina, Benjamin/Cummings, Mento Park, CA, 1986, pp. 381–395.
[22] H. Decker, Personal communication, January 1997.
[23] H. Decker, M. Celma, A slick procedure for integrity checking in deductive databases, in: P. Van Hentenryck (Ed.), Proceedings of ICLP'94, MIT Press, New York, 1994, pp. 456–469.
[24] M. Denecker, D. De Schreye, SLDNFA: An abductive procedure for normal abductive programs, in: K. Apt (Ed.), Proceedings of the International Joint Conference and Symposium on Logic Programming, Washington, 1992.
[25] K. Doets, Levationis laus, J. Logic Comput. 3 (5) (1993) 487–516.
[26] A. Ershov, On Futamura projections (in Japanese), BIT (Japan) 12 (14) (1982) 4–5.
[27] H. Fujita, K. Furukawa, A self-applicable partial evaluator and its use in incremental compilation, New Gener. Comput. 6 (2/3) (1988) 91–118.
[28] Y. Futamura, Partial evaluation of a computation process – an approach to a compiler-compiler, Systems Comput. Controls 2 (5) (1971) 45–50.
[29] J. Gallagher, A system for specialising logic programs, Technical Report TR-91-32, University of Bristol, November 1991.
[30] J. Gallagher, Tutorial on specialisation of logic programs, in: Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New Yok, 1993, pp. 88–98.
[31] J. Gallagher, M. Bruynooghe, The derivation of an algorithm for program specialisation, New Gener. Comput. 9 (3/4) (1991) 305–333.
[32] R. Glück, R. Nakashige, R. Zöchling, Binding-time analysis applied to mathematical algorithms, in: J. Dolezal, J. Fidler (Eds.), Proceedings of the Seventeenth IFIP Conference on System Modelling and Optimization, Prague, Czech Republic, Chapman & Hall, London, 1995.
[33] C.A. Gurr, A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel, Ph.D. Thesis, Department of Computer Science, University of Bristol, January 1994.

[34] C.A. Gurr, Specialising the ground representation in the logic programming language Gödel, in: Y. Deville (Ed.), Logic Program Synthesis and Transformation, Proceedings of LOPSTR'93, Workshops in Computing, Louvain-La-Neuve, Belgium, Springer, Berlin, 1994, pp. 124–140.

[35] P. Hill, J. Gallagher, Meta-programming in logic programming, in: D.M. Gabbay, C.J. Hogger, J.A. Robinson (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, Oxford Science Publications, Oxford University Press, Oxford, 1998, pp. 421–427.

[36] P. Hill, J.W. Lloyd, The Gödel Programming Language, MIT Press, Cambridge, MA, 1994.

[37] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[38] N.D. Jones, A. Mycroft, Stepwise development of operational and denotational semantics for Prolog, in: Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, NJ, IEEE Computer Soc. Press, Silver Spring, MD, pp. 289–298.

[39] J. Jørgensen, M. Leuschel, Efficiently generating efficient generating extensions in Prolog, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation, Schloß Dagstuhl, Lecture Notes in Computer Science, vol. 1110, Springer, Berlin, 1996, pp. 238–262; Extended version as Technical Report CW 221, K.U. Leuven.

[40] H.-P. Ko, M.E. Nadel, Substitution and refutation revisited, in: K. Furukawa (Ed.), Logic Programming: Proceedings of the Eighth International Conference, MIT Press, Cambridge, MA, 1991, pp. 679–692.

[41] J. Komorowksi, A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, Ph.D. Thesis, Linköping Studies in Science and Technology Dissertations 69, Linköping University, Sweden, 1981.

[42] J. Komorowski, An introduction to partial deduction, in: A. Pettorossi (Ed.), Proceedings Meta'92, Lecture Notes in Computer Science, vol. 649, Springer, Berlin, 1992, pp. 49–69.

[43] V. Küchenhoff, On the efficient computation of the difference between consecutive database states, in: C. Delobel, M. Kifer, Y. Masunaga (Eds.), Deductive and Object-Oriented Databases, Second International Conference, Munich, Germany, Springer, Berlin, 1991, pp. 478–502.

[44] A. Lakhotia, L. Sterling, How to control unfolding when specializing interpreters, New Gener. Comput. 8 (1990) 61–70.

[45] S.Y. Lee, T.W. Ling, Improving integrity constraint checking for stratified deductive databases, in: Proceedings of DEXA'94, 1994.

[46] S.Y. Lee, T.W. Ling, Further improvement on integrity constraint checking for stratifiable deductive databases, in: Proceedings of the Twenty-Second VLDB Conference, Bombay, India, 1996.

[47] M. Leuschel, Self-Applicable Partial Evaluation in Prolog, Master's Thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1993.

[48] M. Leuschel, Partial evaluation of the "real thing", in: L. Fribourg, F. Turini (Eds.), Logic Program Synthesis and Transformation – Meta-Programming in Logic, Proceedings of LOPSTR'94 and META'94, Pisa, Italy, Lecture Notes in Computer Science, vol. 883, Springer, Berlin, 1994, pp. 122–137.

[49] M. Leuschel, Prototype partial evaluation system to obtain specialised integrity checks by specialising meta-interpreters, Prototype Compulog II, D 8.3.3, Departement Computerwetenschappen, K.U. Leuven, Belgium, September 1995; obtainable at ftp://ftp.cs.kuleuven.ac.be/pub/compulog/ICLeupel/.

[50] M. Leuschel, The ECCE partial deduction system and the DPPD library of benchmarks; obtainable via http://www.cs.kuleuven.ac.be/~lpai, 1996.

[51] M. Leuschel, Advanced Techniques for Logic Program Specialisation, Ph.D. Thesis, K.U. Leuven, May 1997; accessible via http://www.cs.kuleuven.ac.be/~michael.

[52] M. Leuschel, D. De Schreye, Constrained partial deduction and the preservation of characteristic trees, New Gener. Comput. 16 (1998), to appear.

[53] M. Leuschel, D. De Schreye, Towards creating specialised integrity checks through partial evaluation of meta-interpreters, in: Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, ACM, New York, 1995, pp. 253–263.

[54] M. Leuschel, D. De Schreye, Logic program specialisation: How to be more specific, in: H. Kuchen, S. Swierstra (Eds.), Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96), Aachen, Germany, Lecture Notes in Computer Science, vol. 1140, Springer, Berlin, 1996, pp. 137–151; extended version as Technical Report CW 232, K.U. Leuven.

[55] M. Leuschel, B. Martens, Obtaining specialised update procedures through partial deduction of the ground representation, in: H. Decker, U. Geske, T. Kakas, C. Sakama, D. Seipel, T. Urpi (Eds.), Proceedings of the ICLP'95 Joint Workshop on Deductive Databases and Logic Programming and Abduction in Deductive Databases and Knowledge Based Systems, GMD-Studien Nr. 266, Kanagawa, Japan, June 1995, pp. 81–95.

[56] M. Leuschel, B. Martens, Partial deduction of the ground representation and its application to integrity checking, in: J.W. Lloyd (Ed.), Proceedings of ILPS'95, the International Logic Programming Symposium, Portland, USA, MIT Press, Cambridge, MA, 1995, pp. 495–509; extended version as Technical Report CW 210, K.U. Leuven.

[57] M. Leuschel, B. Martens, D. De Schreye, Controlling generalisation and polyvariance in partial deduction of normal logic programs, ACM Trans. Programming Languages Systems 20 (1) (1998) 208–258.

[58] M. Leuschel, B. Martens, K. Sagonas, Preserving termination of tabled logic programs while unfolding, in: N. Fuchs (Ed.), Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97), Leuven, Belgium, Lecture Notes in Computer Science, Springer, Berlin, to appear.

[59] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1987.

[60] J.W. Lloyd, J.C. Shepherdson, Partial evaluation in logic programming, J. Logic Programming 11 (3/4) (1991) 217–242.

[61] J.W. Lloyd, E.A. Sonenberg, R.W. Topor, Integrity checking in stratified databases, J. Logic Programming 4 (4) (1987) 331–343.

[62] J.W. Lloyd, R.W. Topor, Making PROLOG more expressive, J. Logic Programming 1 (3) (1984) 225–240.

[63] J.W. Lloyd, R.W. Topor, A basis for deductive database systems, J. Logic Programming 2 (1985) 93–109.

[64] B. Martens, Finite unfolding revisited (part II): Focusing on subterms, Technical Report Compulog II, D 8.2.2.b, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1994.

[65] B. Martens, On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming, Ph.D. Thesis, K.U. Leuven, February 1994.

[66] B. Martens, D. De Schreye, Two semantics for definite meta-programs, using the non-ground representation, in: K.R. Apt, F. Turini (Eds.), Meta-logics and Logic Programming, MIT Press, Cambridge, MA, 1995, pp. 57–82.

[67] B. Martens, D. De Schreye, Why untyped non-ground meta-programming is not (much of) a problem, J. Logic Programming 22 (1) (1995) 47–99.

[68] B. Martens, D. De Schreye, Automatic finite unfolding using well-founded measures, J. Logic Programming 28 (2) (1996) 89–146.

[69] B. Martens, D. De Schreye, T. Horváth, Sound and complete partial deduction with unfolding based on well-founded measures, Theoret. Comput. Sci. 122 (1/2) (1994) 97–117.

[70] T. Mogensen, The Application of Partial Evaluation to Ray-Tracing, Master's Thesis, DIKU, University of Copenhagen, Denmark, 1986.

[71] T. Mogensen, A. Bondorf, Logimix: A self-applicable partial evaluator for Prolog, in: K.-K. Lau, T. Clement (Eds.), Logic Program Synthesis and Transformation, Proceedings of LOPSTR'92, Springer, Berlin, 1992, pp. 214–227.

[72] G. Neumann, A simple transformation from Prolog-written metalevel interpreters into compilers and its implementation, in: A. Voronkov (Ed.), Logic Programming, Proceedings of the First and Second Russian Conference on Logic Programming, Lecture Notes in Computer Science, vol. 592, Springer, Berlin, 1991, pp. 349–360.

[73] U. Nilsson, J. Małuszyński, Logic, Programming and Prolog, Wiley, Chichester, 1990.

[74] R. O'Keefe, On the treatment of cuts in Prolog source-level tools, in: Proceedings of the Symposium on Logic Programming, IEEE Press, New York, 1985, pp. 68–72.

[75] S. Owen, Issues in the partial evaluation of meta-interpreters, in: H. Abramson, M. Rogers (Eds.), Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, MIT Press, Cambridge, MA, 1989, pp. 319–339.

[76] S. Prestwich, The PADDY partial deduction system, Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[77] S. Prestwich, An unfold rule for full Prolog, in: K.-K. Lau, T. Clement (Eds.), Logic Program Synthesis and Transformation, Proceedings of LOPSTR'92, Workshops in Computing, University of Manchester, Springer, Berlin, 1992.
[78] S. Prestwich, Online partial deduction of large programs, in: Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New York, 1993, pp. 111–118.
[79] M. Proietti, A. Pettorossi, Semantics preserving transformation rules for Prolog, in: Proceedings of the ACM Symposium on Partial Evaluation and Semantics based Program Manipulation, PEPM'91, Sigplan Notices, vol. 26, No. 9, Yale University, New Haven, USA, 1991, pp. 274–284.
[80] Prolog by BIM 4.0, October 1993.
[81] F. Sadri, R. Kowalski, A theorem-proving approach to database integrity, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Chap. 9, Morgan Kaufmann, Los Altos, CA, 1988, pp. 313–362.
[82] D. Sahlin, An Automatic Partial Evaluator for Full Prolog, Ph.D. Thesis, Swedish Institute of Computer Science, March 1991.
[83] D. Sahlin, Mixtus: An automatic partial evaluator for full Prolog, New Gener. Comput. 12 (1) (1993) 7–51.
[84] R. Seljée, A new method for integrity constraint checking in deductive databases, Data Knowledge Eng. 15 (1995) 63–102.
[85] J.C. Shepherdson, Language and equality theory in logic programming, Technical Report PM-91-02, University of Bristol, 1991.
[86] M. Sperber, Self-applicable online partial evaluation, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation, Schloß Dagstuhl, Lecture Notes in Computer Science, vol. 1110, Springer, Berlin, 1996, pp. 465–480.
[87] L. Sterling, R.D. Beer, Metainterpreters for expert system construction, J. Logic Programming 6 (1/2) (1989) 163–178.
[88] L. Sterling, E. Shapiro, The Art of Prolog, MIT Press, Cambridge, MA, 1986.
[89] A. Takeuchi, K. Furukawa, Partial evaluation of Prolog programs and its application to meta programming, in: H.-J. Kugler (Ed.), Information Processing 86, 1986, pp. 415–420.
[90] M. Wallace, Compiling integrity checking into update procedures, in: J. Mylopoulos, R. Reiter (Eds.), Proceedings of IJCAI, Sydney, Australia, 1991.
[91] D. Weise, R. Conybeare, E. Ruf, S. Seligman, Automatic online partial evaluation, in: Proceedings of the Conference on Functional Programming Languages and Computer Architectures, Harvard University, Lecture Notes in Computer Science, vol. 523, Springer, Berlin, 1991, pp. 165–191.